UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

ANALYSE COMPARATIVE DE LA RÉPARTITION DE L'EFFORT DANS LE CADRE DE PROCESSUS LOGICIELS

présenté par: GERMAIN Éric

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. BOUDREAULT Yves, M.Sc., président
M. ROBILLARD Pierre N., Ph.D., membre et directeur de recherche
M. DESMARAIS Michel, Ph.D., membre

UNIVERSITÉ DE MONTRÉAL

ANALYSE COMPARATIVE DE LA RÉPARTITION DE L'EFFORT DANS LE CADRE DE PROCESSUS LOGICIELS

ÉRIC GERMAIN
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AVRIL 2004

© Éric Germain, 2004.

Aux deux femmes de ma vie

Hughette, ma mère

Marie, ma douce

REMERCIEMENTS

J'aimerais d'abord remercier mon directeur de recherche, Pierre N. Robillard, dont la vision, les conseils, les critiques et les encouragements ont fortement teinté l'ensemble de ma démarche de recherche et de production de ce mémoire.

Je tiens à remercier tout particulièrement Mihaela Dulipovici pour son soutien dans le contexte du cours-projet "Atelier de génie logiciel" et notamment pour son travail méticuleux de validation des jetons d'effort. J'aimerais également remercier Sébastien Cherry, Alexandre Moïse, Christian Robidoux, Martin Robillard et Houcine Skalli pour leurs contributions respectives au succès des Ateliers. Aussi, ce travail de recherche n'aurait pu être possible sans la collaboration des étudiants qui ont participé aux Ateliers.

J'aimerais également remercier :

- Les membres du jury pour leurs commentaires constructifs suite au dépôt initial de ce mémoire ;
- Les réviseurs anonymes pour leurs commentaires pertinents lors de la soumission du premier des deux articles qui constituent le mémoire ;
- Thérèse Gauthier, Shannon Miko et Cynthia Orman pour leurs précieux conseils lors de la révision du mémoire ;
- Sylvie Nadeau pour ses précieux conseils sur le métier d'étudiant aux cycles supérieurs ;

– Robert Roy pour m'avoir incité à me dépasser et pour avoir mis LaTeX sur ma route ;

– mes beaux-parents et mes amis pour leurs encouragements ; et

– les membres de ma famille ainsi que Marc Rochette pour leurs encouragements et leur soutien tout au long de mes études universitaires.

Finalement, un merci tout spécial à ma douce moitié, Marie Brassard, pour ses encouragements, ses conseils, sa patience, sa patience (encore), son soutien de tous les jours et pour avoir révisé le document final. Je t'aime.

RÉSUMÉ

Le domaine des processus de développement de logiciels ne saura être considéré comme mature que lorsque l'élaboration de tels processus pourra être fondée sur une compréhension détaillée et quantitative du comportement des ingénieurs du logiciel. Or, cette vision demeure utopique pour l'instant, car la fondation théorique nécessaire à son accomplissement n'est pas bien ancrée à l'heure actuelle, et beaucoup de travail sera nécessaire avant d'y parvenir. Un des symptômes de la pauvreté relative des connaissances générales dans le domaine est l'absence de consensus face à l'émergence d'approches qui tranchent radicalement avec les enseignements de ce que nous appellerons l'école classique du développement de logiciels. Cette situation découle largement du faible nombre d'études comparant dans leur globalité les différentes approches proposées.

Une des difficultés auxquelles le chercheur risque de faire face dans sa quête d'une vue comparative valable a trait à la mise en place d'un contexte permettant d'effectuer des comparaisons valides. Il est très difficile d'appliquer des processus logiciels divers sur un même ensemble de spécifications sans que des facteurs liés aux projets ne risquent d'interférer avec l'analyse. Cette situation constitue un carcan duquel il faut sortir si l'on veut éventuellement procéder à des études comparatives à grande échelle en milieu industriel. Une façon possible de remédier à ce problème est de cerner les caractéristiques invariables du comportement des ingénieurs du

logiciel par rapport aux conditions de projet, ce qui permettrait éventuellement de procéder à certaines analyses valides dans le contexte de projets variables.

La présente recherche vise à approfondir l'état des connaissances sur la relation entre les caractéristiques des processus et projets de développement de logiciels, définis ou réels, et l'effort déployé par les individus qui développent des systèmes logiciels. Elle a été réalisée en utilisant trois éditions successives d'un cours-projet intitulé « Atelier de génie logiciel », offert chaque année aux étudiants finissants du baccalauréat en génie informatique, concentration génie logiciel, à l'École Polytechnique de Montréal. Deux études distinctes réalisées dans ce cadre sont présentées.

Une première étude, qui a porté sur l'édition 2002 de l'Atelier, a permis de comparer les activités cognitives réalisées par des étudiants lors de travaux de développement de systèmes logiciels réalisés suivant un même ensemble de spécifications, mais selon deux approches distinctes. Elle a nécessité la définition d'une catégorisation originale de ces activités cognitives. Trois des équipes ont utilisé un modèle de processus très fortement inspiré du *Unified Process for Education* (UPEDU), modèle de processus dérivé du *Rational Unified Process* (RUP). Les trois autres équipes ont utilisé un processus construit autour de la méthodologie de programmation extrême (*Extreme Programming* ou « XP »). Selon les observations effectuées, le choix d'une approche particulière a un impact limité sur les activités cognitives réalisées par les équipes étudiées. Cet impact, mesuré en termes relatifs, se manifeste surtout à l'égard des activités d'inspection et de révision des travaux

réalisés.

Une seconde étude, qui a porté sur les éditions 2001, 2002 et 2003, a permis d'analyser les caractéristiques de l'effort déployé par les membres de huit équipes ayant travaillé à partir de trois ensembles de spécifications différents, mais suivant des modèles de processus très fortement inspirés du UPEDU. L'objectif visé par cette étude consistait à repérer des patrons au sein des diagrammes de répartition de l'effort relatif en disciplines à travers le cycle complet de développement d'un système logiciel. Les spécifications soumises, les durées de cycle de développement, les technologies utilisées et les mécanismes de cueillette de l'effort déployé variaient à divers degrés d'une édition à l'autre. Le nombre de participants différait également d'une équipe à l'autre. Une catégorisation en trois disciplines des activités réalisées a été définie de manière à permettre une ségrégation substantielle des activités tout en minimisant l'influence des facteurs liés à l'édition ou à la taille de l'équipe. L'analyse réalisée montre une convergence modérée des proportions d'activités réalisées par chaque équipe sur l'ensemble du cycle de développement. La majorité des projets soumis affichent toutefois une évolution très similaire du type d'activités réalisées au fil du temps. Ainsi, l'effort réalisé suit une progression qui débute par une première phase, où les activités liées à l'ingénierie sont prédominantes. Le poids de ces activités diminue progressivement, ce qui amène une seconde phase d'équilibre entre les activités de programmation et celles de validation et de vérification. Cet équilibre est rompu peu avant la fin du projet en faveur des activités de validation

et de vérification, quoique de manière peu marquée.

En montrant l'indépendance relative du comportement des ingénieurs du logiciel par rapport aux conditions de processus et de projet, le travail de recherche ouvre la voie pour l'application éventuelle des analyses de patrons d'effort vers le milieu industriel. Sur le plan théorique, de telles analyses pourront permettre d'approfondir davantage notre compréhension du comportement des praticiens du développement de logiciels. Sur le plan pratique, elles pourraient constituer un outil puissant d'estimation et de contrôle à la disposition des gestionnaires de projets et des ingénieurs de processus.

ABSTRACT

The software development process field will be considered mature only when the elaboration of such processes can be based on a detailed and quantitative understanding of the behavior of software engineers. Since the theoretical foundations required to achieve this understanding are not currently well defined, maturity remains a utopian vision. A symptom of the paucity of general knowledge on the process is the lack of a consensus with respect to the newly emerging approaches to software development, as they represent a major break from what we may refer to as the 'classical' school of software development. The reason for this lack of consensus is that very few studies have been carried out comparing the various proposed approaches from a global perspective.

One challenge which the researcher may face in the search for a way to usefully compare software development approaches is to find a suitable context in which to make such a comparison. It is extremely difficult, for example, to apply a number of software processes to a single set of specifications and avoid interference in the analysis from project-related factors. Obstacles like these must be removed if comparative studies in industry are to be executed on a large scale. One way to solve this problem is to identify characteristics of the behavior of software engineers which do not vary with project conditions, as this would permit valid analysis in a multi-project context.

This research is aimed at improving the state of knowledge on the relationship between software development process and project characteristics, theoretical or physical, and the effort expended by the developers of software systems. Our study was conducted using three successive editions of the Software Engineering Studio, a project course offered once a year to senior students in the Bachelor of Computer Engineering program (Software Engineering Option) at the École Polytechnique de Montréal. Two separate studies performed in this context are presented.

One of the studies, using the 2002 edition of the Studio, compared the cognitive activities performed by students who were developing software systems based on a single set of specifications, but using two different approaches. This study involved defining a new classification for these activities. Three of the teams used a process very strongly based on the Unified Process for Education (UPEDU), a teaching-oriented process derived from the Rational Unified Process. The other three teams used a process built around the principles of the Extreme Programming (XP) methodology. The results revealed that the approach used has limited impact on the cognitive activities performed by the teams. What impact there was, measured in relative terms, was associated mainly with artifact inspection and review.

The second study, using the 2001, 2002 and 2003 editions of the Studio, evaluated the effort expended by the members of eight teams developing software systems based on three distinct sets of specifications, but using a single process model very

similar to UPEDU. The objective of the study was to identify the patterns that emerged throughout the entire software development cycle in the effort distribution diagrams associated with the disciplines involved. The requirement specifications submitted, the lengths of the development cycles, the technologies used and the effort-recording methods and tools employed all varied to some extent from one edition to another. The number of participants also varied from team to team. The activities were classified into three disciplines in order to permit substantial segregation of the activities, while at the same time minimizing the effect of factors related to the edition used or the size of the team. The results revealed a tendency towards moderate convergence of the relative proportions of the activities performed by the teams. Moreover, these proportions evolved in a similar way over time for a majority of the teams. The effort expended progressed from an initial phase in which the focus was engineering activities, their weight decreasing gradually over the duration of the phase, to a second phase characterized by a balance between coding and validation and verification (V&V) activities, which endured to the end of the project. This balance shifts shortly before the end, however, where there is a stronger emphasis on V&V activities.

By showing the independence of the behavior of software engineers from process and project conditions, this research opens the way to the implementation of effort pattern analysis in industrial settings. From a theoretical perspective, such analyses may help improve our understanding of the behavior of software develop-

ment professionals. From a more practical perspective, they may also constitute powerful tools for software project managers and process engineers.

TABLE DES MATIÈRES

DÉDICACE iv

REMERCIEMENTS v

RÉSUMÉ vii

ABSTRACT xi

TABLE DES MATIÈRES xv

LISTE DES FIGURES xix

LISTE DES SIGLES ET ABBRÉVIATIONS xxi

LISTE DES TABLEAUX xxiii

LISTE DES ANNEXES xxiv

INTRODUCTION 1

CHAPITRE 1 : REVUE CRITIQUE DE LA LITTÉRATURE 6

1.1 Processus logiciels 6

1.1.1 Évolution du domaine 7

1.1.2 Revue des principaux modèles de processus 11

1.2 Méthodologies agiles de développement de logiciels 18

1.2.1 Survol des méthodologies agiles 18
1.2.2 La programmation extrême 20
1.2.3 Les processus d'ingénierie par opposition aux méthodologies agiles 25
1.3 La mesure des processus logiciels 26
1.3.1 L'estimation de l'effort 26
1.3.2 La mesure de l'effort 29
1.4 Le contrôle des processus 31

CHAPITRE 2 : ORGANISATION GÉNÉRALE DU TRAVAIL 34

CHAPITRE 3: ENGINEERING-BASED PROCESSES AND AGILE METHODOLOGIES FOR SOFTWARE DEVELOPMENT: A COMPARATIVE CASE STUDY 38
3.1 Introduction 39
3.2 The Software Engineering Studio 43
3.3 Cognitive activity classification 50
3.4 Analysis of cognitive activities performed 53
3.4.1 Raw-data analysis 53
3.4.2 Aggregate analysis 56
3.4.3 Inter-process analysis 58
3.5 Concluding remarks 60

3.5.1 Study limitations 60
3.5.2 Observations on the methodology 63
3.5.3 Participants' behaviour 65
3.5.4 Teaching software development 66

CHAPITRE 4: MEASURING RELATIVE EFFORT PROGRESSION PATHS IN THE DISCIPLINE SPACE OF THE SOFTWARE ENGINEERING PROCESS 68

4.1 Introduction 68
4.2 The Software Engineering Studio 72
4.3 Team-projects under study 73
4.4 Overview of disciplines 75
4.5 Study of effort distribution invariance relative to project factors . . 76
4.5.1 Regular and coherent project progressions 78
4.5.2 Coherent progressions with tilt towards coding 83
4.5.3 Incoherent progressions 84
4.6 Non-cumulative perspective on regular coherent progressions 85
4.7 Analysis 90
4.7.1 Coherence and regularity 90
4.7.2 Development cycle and key milestones for regular and coherent progressions 90
4.7.3 Impacts on effort estimation and process control 92

4.7.4 Study limitations 93

4.7.5 Effort distribution: does it make sense? 94

4.8 Conclusion 94

CHAPITRE 5 : DISCUSSION GÉNÉRALE 97

5.1 Calibration des charges de travail 97

5.2 La classification en disciplines 98

CONCLUSION 103

BIBLIOGRAPHIE 108

LISTE DES FIGURES

1.1 Modèle conceptuel fondamental du RUP (traduit de Robillard, Kruchten et d'Astous 2003) 13
1.2 Modèle bidimensionnel de processus et de cycle de vie du UPEDU (traduit de École Polytechnique de Montréal 2002) 14
1.3 Définition des jalons au cours du cycle de développement (traduit de École Polytechnique de Montréal 2002) 16
1.4 Évolution comparée des coûts du changement au fil du cycle de développement, selon les idées de Beck (1999) 21
1.5 Modèle simplifié du cycle de développement et de maintenance en programmation extrême 23

3.1 Screenshot from a completed product 45
3.2 Software development schedule (in working days) 46
3.3 Effort distribution by activity, by team 54
3.4 Aggregate effort distribution by cognitive activity 56
3.5 Aggregate effort by cognitive activity, by process (%) 57
3.6 Weight index by cognitive activity 58
3.7 Weight index by category 60

4.1 Distribution of effort among team-projects 74

4.2 Relative effort distribution by category, for all team-projects, sorted by increasing total effort 76

4.3 2D plot of total team-project effort ratios relative to the three poles 77

4.4 2D plot of effort-mix progression for team-project 2001A 79

4.5 2D plot of effort-mix progression for team-project 2001C 79

4.6 2D plot of effort-mix progression for team-project 2002B 80

4.7 2D plot of effort-mix progression for team-project 2003A 80

4.8 2D plot of effort-mix progression for team-project 2001B 81

4.9 2D plot of effort-mix progression for team-project 2003B 81

4.10 2D plot of effort-mix progression for team-project 2002A 82

4.11 2D plot of effort-mix progression for team-project 2003C 82

4.12 Non-cumulative effort for team-projects with regular and coherent progressions as a function of project advancement 86

4.13 Non-cumulative effort for team-projects with irregular but coherent progressions as a function of project advancement 87

4.14 Non-cumulative effort for team-projects with incoherent progressions as a function of project advancement 89

LISTE DES SIGLES ET ABBRÉVIATIONS

2D	Bi-dimensional
Cn	Itération de construction n
CMM	Capability Maturity Model
COCOMO	Constructive Cost Model
En	Itération d'élaboration n
EQ	Modèle équivalent
EQM	Modèle équivalent, itération de maintenance
EQn	Modèle équivalent, itération n
IEEE	Institute of Electrical and Electronics Engineers
It.	Itération
NSERC	Natural Sciences and Engineering Research Council of Canada
OPEN	Object-oriented Process, Environment and Notation
PAI	Process Activity Index
PSEE	Process-centered Software Engineering Environment
PSP	Personal Software Process
RUP	Rational Unified Process
SLIM	Software Lifecycle Management
SPCC	Software Project Control Center

Tn	Itération de transition n
TSP	Team Software Process
UP	Unified Process for Education
UPM	Unified Process for Education, itération de maintenance
UPn	Unified Process for Education, équipe n (si n = A, B ou C), ou itération n (si n est numérique)
UPEDU	Unified Process for Education
V&V	Validation and Verification
XP	Extreme Programming
XPM	Extreme Programming, itération de maintenance
XPn	Extreme Programming, équipe n (si n = A, B ou C), ou itération n (si n est numérique)

LISTE DES TABLEAUX

3.1 Cognitive activity classification 51

5.1 Un exemple de classification en disciplines, basée sur les disciplines originales du UPEDU 102

LISTE DES ANNEXES

Annexe I : Calculation of the 2D plots of effort 119

Annexe II : Contenu du support CD fourni 122

INTRODUCTION

> In software engineering there is no theory. (. . .) It's all arm flapping and intuition. I believe that a theory of software evolution could eventually translate into a theory of software engineering. Either that or it will come very close. It will lay the foundation for a wider theory of software evolution.(Manny Lehman, cité dans Williams 2002)

La capacité de définir de manière détaillée et en des termes quantitatifs le comportement des développeurs de logiciels serait susceptible d'ouvrir la porte à l'avènement de méthodes de développement automatisé, via l'implantation d'environnements de génie logiciel centrés sur les processus (*Process-centered Software Engineering Environments* ou PSEE) (Cugola et Ghezzi 1998). Toutefois, cette vision demeure utopique pour l'instant, car la fondation théorique nécessaire à son accomplissement n'est pas bien ancrée à l'heure actuelle. Ainsi, les connaissances sur les déterminants des caractéristiques d'un processus logiciel réel, c'est-à-dire tel qu'appliqué, sont limitées. Or, le développement de modèles valables, qu'ils soient simples ou complexes, ne peut être réalisé avec assurance qu'avec une fondation théorique minimale.

Un des symptômes de la pauvreté relative des connaissances générales dans le domaine est l'absence de consensus face à l'émergence d'approches qui tranchent radicalement avec les enseignements de ce que nous appellerons l'école classique

du développement de logiciels. Ainsi, les dernières années ont été marquées par l'émergence de méthodologies dites agiles (Cockburn 2002) et, notamment, de la programmation extrême (*Extreme Programming* ou XP) (Beck 1999). Cette dernière prône essentiellement le relâchement des efforts spécifiques en matière de planification, d'analyse et de conception, à la faveur d'une stratégie de réduction des coûts du changement et de retardement des décisions principales en matière de requis et de conception. Les processus traditionnels (voir par exemple Kruchten 2000), au contraire, préconisent l'application rigoureuse de pratiques d'ingénierie et de gestion, ainsi que la production de plusieurs artéfacts. Il est intéressant de noter que, malgré le débat sur la question (voir par exemple Antón 2003; Wells 2003), les arguments soulevés des deux côtés ramènent souvent soit à des aspects théoriques non vérifiés, soit à des expériences personnelles difficilement généralisables. En outre, les études comparatives documentées entre les deux approches sont rares.

Par ailleurs, le chercheur désireux de comparer deux ou plusieurs approches doit mettre en place un contexte permettant de rendre de telles comparaisons valides. Or, il est très difficile d'appliquer des processus logiciels distincts sur un même ensemble de spécifications sans que des facteurs liés aux projets ne risquent d'interférer avec l'analyse. Cette situation constitue un carcan duquel il faut sortir si l'on veut éventuellement procéder à des études comparatives à grande échelle en milieu industriel. Une façon possible de remédier à ce problème est de cerner les caractéristiques invariables du comportement des ingénieurs du logiciel par rapport

aux conditions de projet, ce qui permettrait éventuellement de procéder à certaines analyses valides dans le contexte de projets variables.

Le département de génie informatique de l'École Polytechnique de Montréal utilise depuis plusieurs années une plate-forme académique afin d'approfondir l'état des connaissances sur les processus logiciels. Le cours-projet « Atelier de génie logiciel », offert principalement aux étudiants finissants du baccalauréat en génie informatique, concentration « génie logiciel », constitue non seulement une opportunité de formation pratique mais également un environnement de recherche permettant d'observer et d'analyser des processus réels de développement appliqués à la réalisation de systèmes complets par les étudiants. L'Atelier a servi de base à la réalisation de quelques études empiriques (Germain, Dulipovici et Robillard 2002a; Germain, Robillard et Dulipovici 2002b; Slavich 2000) couvrant une seule édition à la fois et portant principalement sur la comparaison de l'effort déployé par des équipes devant produire des implantations distinctes, conformes à des spécifications communes.

La recherche présentée dans le cadre de ce mémoire a été entreprise dans le but d'approfondir l'état des connaissances sur la relation entre les caractéristiques des processus et projets de développement de logiciels, définis ou réels, et le comportement des individus qui développent des systèmes logiciels. Elle est exploratoire et repose sur les questions suivantes :

– Quel est l'impact du choix d'une approche de développement sur le compor-

tement des ingénieurs du logiciel ?

- Existe-t-il des invariants ou des patrons d'effort qui caractérisent le comportement type des ingénieurs du logiciel ?
- Ces invariants et patrons d'effort transcendent-ils les caractéristiques de projets et de processus ?

Le travail présenté a été réalisé en utilisant trois éditions successives de l'Atelier décrit précédemment. Deux études distinctes réalisées dans ce cadre sont présentées :

- Une première étude, qui a porté sur l'édition 2002 de l'Atelier, a permis de comparer les activités cognitives réalisées par des étudiants sur un même ensemble de spécifications mais selon deux approches distinctes de développement.
- Une seconde étude, qui a porté sur les éditions 2001, 2002 et 2003, a permis d'analyser les caractéristiques de l'effort déployé par les membres de huit équipes ayant travaillé à partir de trois ensembles de spécifications différents mais suivant des modèles de processus à peu près identiques.

Le premier chapitre présente une revue de la littérature pertinente au domaine des processus et méthodologies de développement de logiciels, qui constitue l'objet principal du présent travail de recherche. Suit une discussion sur la mesure des processus, afin de sensibiliser le lecteur aux problématiques liées à la détection d'informations pertinentes au sein d'un processus logiciel réel. Finalement, on effleure

le domaine du contrôle des processus, susceptible de bénéficier des résultats du présent travail.

Le chapitre 2 aborde l'organisation du présent travail. Le chapitre 3 reproduit le premier des deux articles présentés dans le cadre du mémoire, alors que le second article est présenté au chapitre 4. Enfin, le chapitre 5 présente une discussion globale de la recherche, complémentaire aux conclusions énoncées à l'intérieur des articles.

CHAPITRE 1

REVUE CRITIQUE DE LA LITTÉRATURE

1.1 Processus logiciels

Le présent chapitre constitue une revue critique de la littérature pertinente aux nombreux aspects touchés par ce travail de recherche. Les deux premières sections du chapitre traiteront de l'objet d'étude principal de ce travail, soit les processus et méthodologies de développement de logiciels. Dans la première section, l'état du domaine des processus logiciels sera d'abord décrit, puis trois modèles de processus courants seront présentés. La seconde section traitera des méthodologies de développement de logiciels et, en particulier, de la méthodologie dite de programmation extrême. Par la suite, la mesure des processus logiciels et de l'effort sera abordée dans la perspective où tout travail d'analyse empirique comme celui-ci doit tenir compte du contexte de mesure. Finalement, le domaine du contrôle des processus et de l'estimation en cours de projet sera effleuré. Bien que ces derniers concepts ne soient pas couverts dans le cadre des travaux réalisés, ils constituent une avenue de choix pour l'exploration des conclusions de la recherche effectuée.

1.1.1 Évolution du domaine

Fugetta (2000) offre une perspective historique de l'évolution des processus logiciels. La période écoulée entre les années 1960 et 1980 a été marquée par le développement parallèle de langages de programmation structurée, de méthodes et principes de conception, ainsi que de cycles de développement de logiciels (aussi appelés « cycles de vie », bien que cette appellation puisse porter à confusion comme nous le verrons plus loin). Le cycle de développement constitue une notion clé dans la définition globale des tâches menant à la réalisation d'un système logiciel de qualité ; il définit les diverses étapes de l'évolution d'un produit logiciel, ainsi que les principes directeurs de l'exécution de ces étapes.

Toutefois, le cycle de développement ne saurait prescrire le détail des activités, structures organisationnelles, outils, procédures, politiques et contraintes. Ces éléments font plutôt partie du processus logiciel, ou processus de développement logiciel. Un processus logiciel peut être défini comme étant l'ensemble des politiques, structures organisationnelles, technologies, procédures et artéfacts nécessaires à la conception, au développement, au déploiement et à la maintenance d'un produit logiciel (Fugetta 2000).

Notons au passage que la distinction entre processus et cycle de développement n'est pas toujours claire, particulièrement lorsqu'il est question de ce qui fut connu plus tard sous le nom de modèle de cycle de vie en cascade, introduit par Royce (1970) ; ce modèle, le premier du genre, prévoit implicitement la juxtaposition pure

et simple des deux concepts.

Osterweil et Lehman ont, par l'opposition de leurs vues sur le sujet, lancé une nouvelle ère en matière de recherche sur les processus logiciels. Ainsi, Osterweil (1987) considère qu'un des problèmes principaux empêchant l'assimilation de la notion de processus logiciel à l'application de procédures ou recettes pour la fabrication d'avions ou d'automobiles est l'absence de gabarits à partir desquels un processus logiciel appliqué à une situation spécifique pourrait être « instancié ». L'approche d'Osterweil consiste à définir de tels gabarits de la manière la plus rigoureuse possible, soit à travers la rédaction de « programmes » de processus utilisant des langages aussi formels que les langages de programmation informatique. Cette approche met de l'avant une recherche axée vers la création d'un langage de programmation de processus et d'un système de compilation et d'interprétation des programmes rédigés.

Cugola et Ghezzi (1998), quant à eux, croient que le concept d'environnement de génie logiciel centré sur les processus (PSEE), qui découle naturellement des théories d'Osterweil, constitue la voie privilégiée de la recherche sur les processus du génie logiciel et ce, en dépit de l'impact très faible sur le terrain jusqu'ici de ces environnements et des langages de modélisation associés. Les auteurs prétendent notamment que cette situation défavorable découle du peu de flexibilité des environnements développés jusqu'ici, et non de problèmes structurels. Ainsi, les environnements développés dans le futur auront un bien meilleur impact s'ils

tolèrent, par exemple, la déviation explicite du processus par les développeurs.

À l'inverse, Lehman (1987) prétend que la programmation de processus telle que définie par Osterweil ne peut que détourner la collectivité du génie logiciel des problèmes réels auxquels la discipline est confrontée, plutôt que de contribuer à la résolution de ceux-ci. Pis encore, au lieu de contribuer à clarifier la nature des processus réels, elle crée plutôt l'illusion du progrès. Ainsi, quoique les programmes de processus soient théoriquement utiles pour l'expression formelle des processus, ils ne sont d'aucune utilité pour l'amélioration de notre compréhension des dynamiques obscures qui composent, encore aujourd'hui, la plus grande partie de la discipline. Du même souffle, Lehman mentionne :

> For applications (commonly termed "programming-in-the-large"), which provide the real challenge for software engineering as distinct from programming methodology, models of the application as a whole or of many of its parts do not, in general, exist; there is no theory of program development, there is no global and formalisable development procedure, at best there is only an abstract process model.

Dix ans plus tard, il déclare : « Eventually it may be possible to develop generic models but that lies in the distant future (Lehman 1997). »

Les vues de Lehman trouvent écho dans le développement récent de la pratique du génie logiciel. Ainsi, Robillard, Kruchten et d'Astous (2003, p. 41) mentionnent que le développement de logiciels comporte une forte composante opportuniste,

c'est-à-dire qu'une démarche exploratoire non planifiée est très souvent utilisée afin d'identifier les éléments d'information manquants qui auraient permis d'appliquer une démarche plus systématique. De plus, une progression d'un mode de développement systématique vers un mode opportuniste est généralement observée au fil de l'évolution d'un projet. La présence d'activités opportunistes semble difficilement compatible avec le concept de programmation de processus d'Osterweil. D'une manière similaire, Brooks trouve dans le processus de construction de logiciels une dimension créative cruciale : « Great designs come from great designers. Software construction is a *creative*[1] process. Sound methodology can empower and liberate the creative mind ; it cannot inflame or inspire the drudge (Brooks 1987). »

Dans un tel contexte, nous sommes en droit de nous demander comment la programmation de processus réussirait à compenser pour la faiblesse possible, voire même probable, des capacités de design d'une équipe de développement, dans le contexte de l'absence de compréhension des facteurs qui, justement, font qu'un concepteur sera meilleur qu'un autre.

Le présent travail repose explicitement sur l'hypothèse selon laquelle la recherche sur les processus du génie logiciel doit viser un objectif d'élaboration de modèles permettant de mieux comprendre les facteurs qui les régissent, notamment les facteurs humains. Nous croyons que cette recherche contribue à faire progresser l'état des connaissances dans les domaines décrits par Lehman.

[1] En italique dans le texte

1.1.2 Revue des principaux modèles de processus

En l'absence de modèles quantitatifs formels du développement de logiciels, modèles qui auraient permis de développer un ou plusieurs modèles quantitatifs génériques de processus logiciels, certains intervenants des milieux académiques et industriels ont établi divers modèles de processus qui reposent à la fois sur certains principes issus de la recherche actuelle ou passée et sur des heuristiques découlant d'années d'expérience individuelle ou collective en matière de développement de logiciels. Nous survolerons ici, de manière non exhaustive, quelques-uns de ces modèles.

Il incombe cependant, avant de réaliser ce survol, de faire la distinction entre les types de modèles de processus suivants :

- les modèles prédictifs, qui ont pour fonction de soutenir une démarche de prédiction ou d'estimation ;
- les modèles explicatifs, qui ont pour fonction d'expliquer le comportement réel des participants au processus ; et
- les modèles prescriptifs qui, à l'inverse des modèles explicatifs notamment, visent à indiquer aux participants les activités qui doivent être réalisées dans le cadre de leurs fonctions.

La présente recherche s'intéresse avant tout aux modèles utilisés généralement en milieu industriel, qui sont généralement prescriptifs. Par conséquent, nous ne tiendrons pas compte ici des autres types de modèles.

1.1.2.1 Le *Rational Unified Process*

Le *Rational Unified Process* (RUP) (Kruchten 2000) a été conçu par Rational Software Corporation (acquise depuis par IBM). Il s'agit d'un modèle de processus itératif et incrémental[2], centré sur l'architecture du logiciel plutôt que sur le code (Kruchten 2000). Kruchten définit le RUP comme étant un « produit-processus » (*process product*). Le RUP se présente sous la forme d'un système logiciel comprenant essentiellement un site Web interactif ainsi que divers outils comme des mentors, des gabarits et des exemples d'artéfacts. La figure 1.1 illustre le modèle conceptuel fondamental du RUP. Le RUP est constitué de trois éléments fondamentaux : le rôle, l'artéfact et l'activité. Un rôle constitue une définition des diverses responsabilités pouvant être attribuées à un membre d'une équipe de développement (Robillard, Kruchten et d'Astous 2003). Ainsi, chacune des activités peut être réalisée par un et un seul rôle, et chacun des artéfacts est sous la responsabilité d'un seul rôle. Une activité consomme un certain nombre d'artéfacts intrants et produit également un certain nombre d'artéfacts extrants. Un artéfact peut être consommé ou produit par un nombre indéterminé d'activités.

Le modèle conceptuel fondamental constitue en réalité un composant du métamodèle d'ingénierie de processus logiciel du Object Management Group (2002) et se situe ainsi au niveau M2 de la hiérarchie des niveaux de modélisation de

[2]L'évolution du concept de développement itératif et incrémental est décrite dans Larman et Basili (2003)

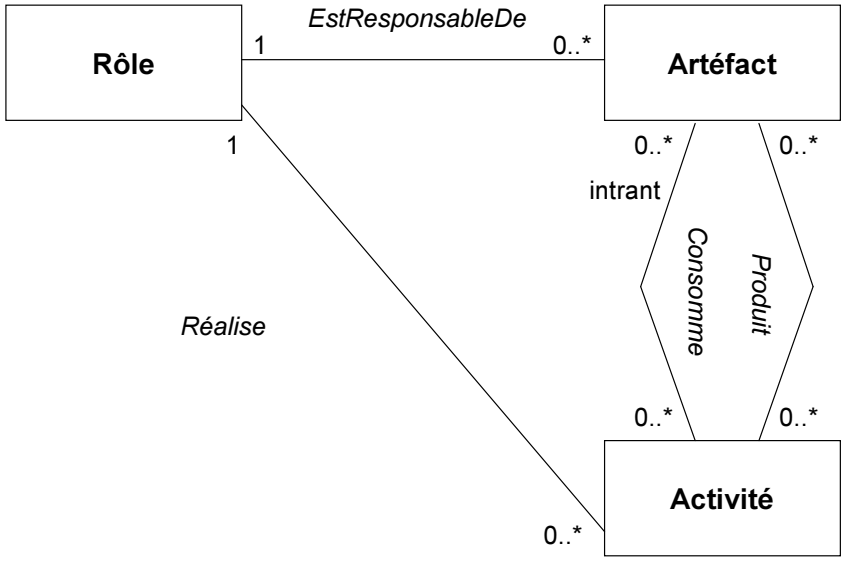

Figure 1.1 – Modèle conceptuel fondamental du RUP (traduit de Robillard, Kruchten et d'Astous 2003)

cette organisation. Il peut s'appliquer à d'autres modèles de processus que le RUP. Il est donc important de noter que l'indétermination des multiplicités du modèle (nombre d'artéfacts intrants et extrants par activité, nombre d'activités consommatrices et productrices par artéfact, nombre d'activités et d'artéfacts par rôle) existe uniquement sur le plan du métamodèle. En pratique, le modèle de processus définit exactement les rôles, activités et artéfacts proposés ainsi que les relations entre ceux-ci.

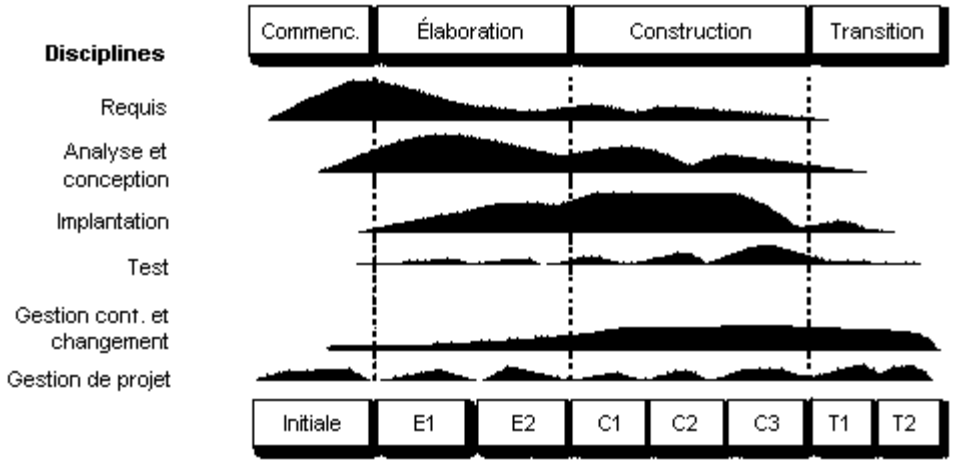

Figure 1.2 – Modèle bidimensionnel de processus et de cycle de vie du UPEDU (traduit de École Polytechnique de Montréal 2002)

1.1.2.2 Le *Unified Process for Education (UPEDU)*

Le UPEDU (École Polytechnique de Montréal 2002) est un modèle de processus dérivé du Rational Unified Process. Il a été mis sur pied par l'École Polytechnique en collaboration avec Rational Software Corporation. Il a été décrit en détail dans Robillard, Kruchten et d'Astous (2001, 2003). Les caractéristiques du RUP telles que mentionnées à la section précédente s'appliquent également au UPEDU, sauf mention contraire. La figure 1.2 illustre le modèle bidimensionnel de processus et de cycle de vie du UPEDU, qui constitue un sous-ensemble du modèle correspondant

à l'intérieur du Rational Software Process. En abscisse figurent les quatre phases qui composent le cycle de vie ; ces phases sont les mêmes que pour le RUP. Chacune des phases peut être subdivisée en un certain nombre d'itérations, suivant le détail du processus retenu ou à la discrétion du gestionnaire de projet.

- La phase de commencement (*inception*) a pour but de permettre à tous les intervenants de s'entendre sur les objectifs du cycle de vie du logiciel. Elle est plus importante pour les projets de développement de nouveaux logiciels que pour les projets de maintenance.
- La phase d'élaboration vise à établir les requis du système à développer et à établir en détail l'architecture de celui-ci.
- La phase de construction vise à revoir les aspects perfectibles des requis et à produire le système selon l'architecture retenue.
- La phase de transition comprend la réalisation des tests de validation et des derniers ajustements ainsi que la mise en disponibilité du logiciel aux usagers.

En ordonnée de la figure 1.2 figurent les six disciplines qui composent le modèle. Chaque discipline constitue un regroupement des activités susceptibles d'être réalisées afin de produire un ensemble spécifique d'artéfacts (École Polytechnique de Montréal 2002). Une activité constitue la plus petite unité de travail mesurable effectuée par un membre de l'équipe de développement, alors qu'un artéfact est défini comme étant un élément d'information produit dans le cadre de l'exécution du processus logiciel (Robillard, Kruchten et d'Astous 2003). Les six courbes de la

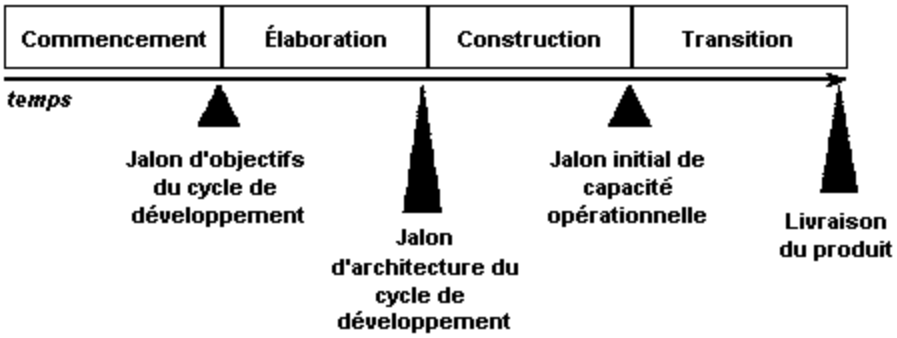

Figure 1.3 – Définition des jalons au cours du cycle de développement (traduit de École Polytechnique de Montréal 2002)

figure 1.2 illustrent l'effort déployé au sein de chaque discipline au fil d'un projet. La distribution de l'effort telle qu'illustrée est purement arbitraire et ne présuppose aucunement de l'allure des efforts déployés au cours d'un projet réel.

La figure 1.3 illustre les jalons associés à chacune des phases du cycle de développement. L'atteinte d'un jalon entraîne l'évaluation de l'atteinte des objectifs de la phase correspondante. La phase subséquente peut être entreprise uniquement dans le cas où ces objectifs sont atteints.

- Le jalon d'objectifs du cycle de développement vise à évaluer les objectifs du projet entrepris.
- Le jalon d'architecture du cycle de développement touche l'évaluation de la portée et des objectifs détaillés du système, de l'architecture choisie et des risques principaux.

- Le jalon initial de capacité opérationnelle vise à s'assurer que toutes les fonctionnalités ont été développées, que tous les tests « alpha » ont été réalisés, qu'un manuel d'utilisation a été préparé et qu'une description de la livraison sous étude a été rédigée.
- Le jalon de livraison du produit correspond à la réalisation réussie de la revue d'acceptation de projet.

Le *Object-oriented Process, Environment and Notation* (OPEN) (Open Consortium 2002; Henderson-Sellers 2000) est un modèle de processus du domaine public qui, comme le RUP, est itératif, incrémental et basé sur le paradigme orienté-objet. Ce modèle de processus a été conçu de manière à pouvoir soutenir de multiples modèles de cycles de développement, comme le modèle en cascade ou encore le modèle en spirale (Boehm 1988). Le concept de phase n'a pas été inclus dans ce modèle. On y définit plutôt les termes suivants :

- Une activité est un ensemble de tâches qui se succèdent au moyen de transitions formelles (pré-conditions et post-conditions) et qui, collectivement, mènent à la réalisation d'un objectif à long terme et débouchent sur un ensemble de produits de travail.
- Une tâche constitue la plus petite unité de travail pouvant être évaluée.
- Une technique englobe l'ensemble du savoir-faire requis pour la réalisation des tâches, la relation entre tâches et techniques étant définie par le biais d'une matrice d'adaptation.

– La notion de produit de travail (*work product*) correspond approximativement à la notion d'artéfact du RUP.

Henderson-Sellers (2000) prétend que « The beauty of the OPEN framework is that it does not lay down the law on what you shall and shan't do. » et qu'il peut, et doit, être adapté aux organismes et projets visés. Quant au RUP, même s'il a été conçu de manière à pouvoir être appliqué intégralement, on pourrait l'adapter également (Kruchten 2000). Il est par conséquent difficile d'établir une comparaison entre les deux modèles en présence sur cette base, comparaison qui déborde du cadre de cet ouvrage.

1.2 Méthodologies agiles de développement de logiciels

L'un des deux articles intégrés dans le présent ouvrage touche directement l'utilisation de la méthodologie de programmation extrême (*extreme programming* ou XP). Cette méthodologie fait partie de la grande famille dite des méthodologies agiles. Nous présenterons d'abord les caractéristiques générales de cette famille puis nous présenterons plus en détail la méthodologie elle-même.

1.2.1 Survol des méthodologies agiles

Les méthodologies dites agiles ont pour but de soutenir le développement de logiciels d'une manière moins rigide que par l'utilisation de processus formels. Le Manifeste du développement agile (Agile Alliance 2003) mentionne les valeurs vé-

hiculées par cette philosophie du développement de logiciels :

- priorité des individus et des interactions sur les processus et les outils ;
- priorité du logiciel fonctionnel sur l'exhaustivité de la documentation ;
- priorité de la collaboration client-fournisseur sur la négociation contractuelle ;
- priorité de la réponse au changement sur le suivi de plans bien définis.

Bien que le Manifeste ait été rédigé principalement en réaction aux tendances contemporaines en matière de processus logiciels, il n'en demeure pas moins que la nomenclature « processus » et « méthodologies agiles » ne constitue pas une dichotomie comme telle. Ainsi, Glass (2001) énumère les forces et les faiblesses respectives des deux approches et plaide pour la combinaison pure et simple des meilleures idées. Kruchten (2001), quant à lui, prétend que la structure du RUP permet l'intégration des caractéristiques de flexibilité recherchées.

Les méthodologies agiles sont avant tout des méthodologies indépendantes qui existaient déjà pour la plupart au moment de la rédaction du Manifeste, plutôt qu'une famille de méthodologies dérivées d'une philosophie originelle. Pour cette raison, nous considérons qu'il convient de passer directement à l'étude de la méthodologie dont il est question dans ce travail de recherche, soit la méthodologie de programmation extrême.

1.2.2 La programmation extrême

La programmation extrême ne constitue pas un modèle de processus à proprement parler mais plutôt une méthodologie reposant sur un ensemble de treize pratiques. Cette méthodologie, décrite en détail dans Beck (1999, 2000), repose sur le rejet du postulat de l'augmentation exponentielle du coût des modifications à un logiciel au fur et à mesure qu'elles se déroulent tard dans le cycle de développement (Pressman 2000). L'application des pratiques de la programmation extrême, et en particulier l'application d'un cycle incrémental de développement suivant des itérations extrêmement courtes, ponctuées de livraisons intermédiaires obligatoires, se veut le mode d'emploi vers l'obtention d'une courbe croissante mais plafonnée des coûts de modification du logiciel en fonction du moment où elles sont amenées (voir figure 1.4). La relation entre les courbes de coûts pour chaque cas peut s'exprimer mathématiquement comme suit. Soient t le temps écoulé depuis le début de la réalisation d'un projet ; c le coût d'un changement réalisé au temps t ; et n une constante quelconque. Pour $n, t, c \in \mathbb{R}; n > 1; t \geq 0$:

– Processus conventionnel :

$$c(t) = t^n \tag{1.1}$$

– Programmation extrême :

$$c(t) = t^{1/n} \tag{1.2}$$

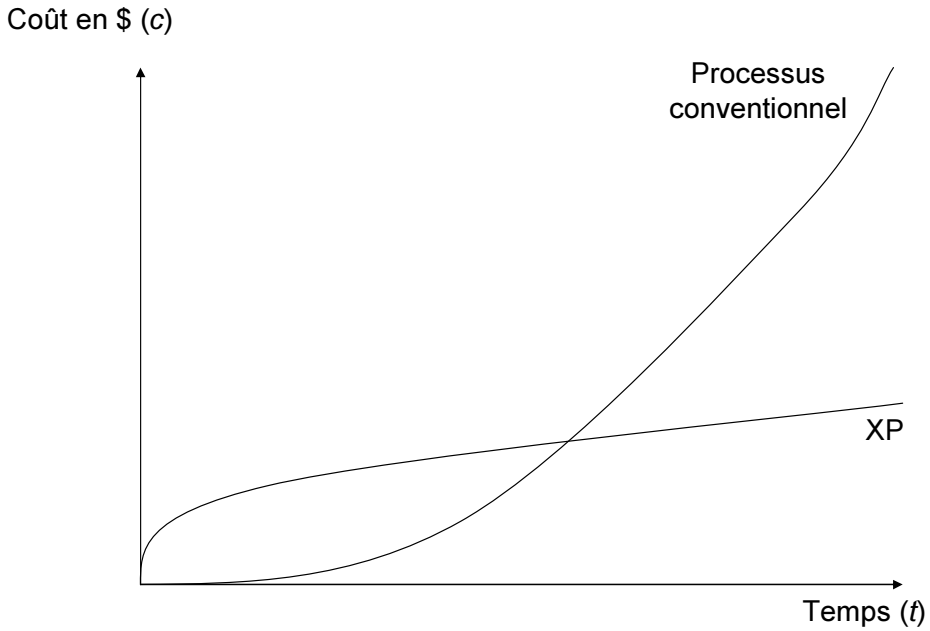

Figure 1.4 – Évolution comparée des coûts du changement au fil du cycle de développement, selon les idées de Beck (1999)

Les conséquences sur la stratégie de développement découlant de l'utilisation de l'une ou l'autre de ces courbes sont cruciales. L'équation 1.1 mène directement à une stratégie visant à minimiser les coûts du changement par la prise des décisions de fond le plus tôt possible dans le projet, notamment sur les volets des requis et de l'architecture, et par le gel de celles-ci. De manière contrastée, dans le contexte de l'équation 1.2, le retardement de la prise des décisions de fond comporte un impact beaucoup moindre sur la profitabilité du projet. Il s'ensuit que, selon la théorie de la programmation extrême (Beck 1999), le gestionnaire de projet doit se rabattre,

dans sa quête d'une plus grande économie dans la réalisation du projet, sur les méthodes classiques de la maximisation de la valeur économique des projets (Lutz 1992). Une telle approche passe par l'application de la méthodologie de planification des travaux suivante.

- Il s'agit d'abord de percevoir le projet non pas comme une entité unique mais plutôt comme une succession de mini-projets appelés « livraisons », fondés successivement les uns sur les autres. Ainsi, la version la plus récente du système logiciel est livrée au client à la fin de chaque mini-projet.
- Il faut ensuite procéder à l'ordonnancement des travaux au niveau macroscopique, ainsi qu'à la planification de la première livraison afin de maximiser la valeur de celle-ci dans la perspective du client. Les caractéristiques qui ne contribuent pas à cet objectif sont exclues de la livraison.
- Une fois la livraison effectuée, il s'agit d'en facturer immédiatement le contenu et de passer à la livraison suivante.

Cette approche par livraisons fréquentes doit permettre de maximiser la valeur du projet par l'application des trois tactiques suivantes :

- Les dépenses surviennent plus tard car l'implantation des caractéristiques non essentielles à la qualité d'une livraison est constamment retardée. Ce retardement contribue à réduire le coût net du projet. Pour cette même raison, les caractéristiques initialement souhaitées, mais pour lesquelles l'intérêt du client diminue au fil du projet, finissent par ne jamais être implantées, ce qui

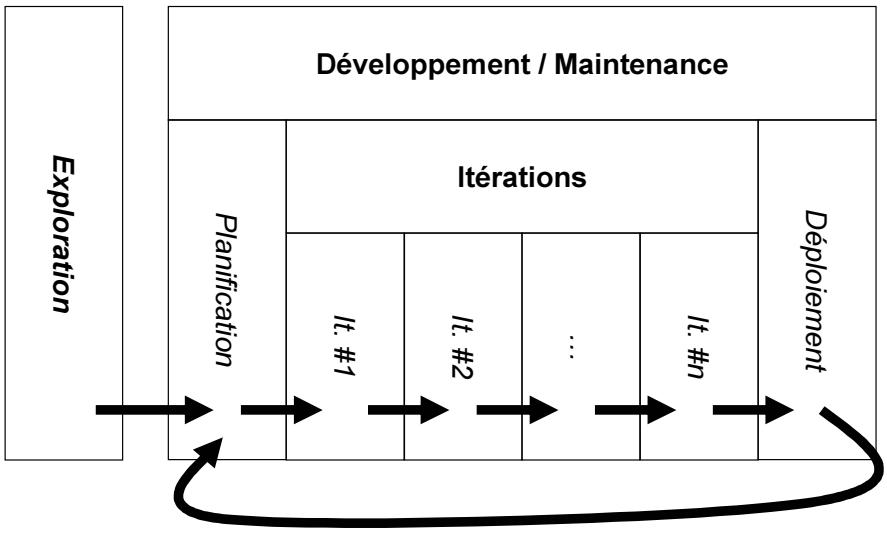

Figure 1.5 – Modèle simplifié du cycle de développement et de maintenance en programmation extrême

a des effets directs sur le coût du projet.

– L'obtention de revenus est plus rapide, étant donné la facturation à chaque livraison, ce qui diminue le risque financier associé au projet, allège les besoins en liquidités et augmente la valeur nette du projet.
– Le risque d'échec est réduit grâce à la validation constante de la valeur du projet par le client.

La figure 1.5 illustre un modèle simplifié du cycle de développement et de maintenance en programmation extrême, basé sur Ambler (2002); Beck (2000). Le cycle de développement ou de maintenance constitue le cœur du modèle. Il englobe les phases de planification, de développement itératif et de déploiement. Chaque exécu-

tion du cycle mène à une livraison du logiciel. Le cycle assurant la livraison initiale est appelé « cycle de développement », alors que ceux débouchant sur les livraisons suivantes sont appelés « cycles de maintenance ».

La phase d'exploration amorce le projet et permet à l'équipe de développement de s'entendre avec le client sur le contenu souhaité du logiciel et d'évaluer avec suffisamment de précision les coûts et délais associés à l'envergure de ce contenu. Sa portée peut s'étendre à l'ensemble de la durée du projet (Ambler 2002) ou simplement à la livraison à venir (Beck 2000).

La phase de planification est réalisée strictement dans la perspective de la livraison à laquelle elle se rattache. Elle permet à l'équipe de développement de s'entendre avec le client sur le contenu de la première livraison ainsi que sur une date de livraison.

La phase de développement itératif est conforme aux principes du développement incrémental (Mills 1980). Ainsi, le logiciel est développé graduellement au cours des diverses itérations planifiées au début de la phase. Chaque itération est d'une durée d'une à quatre semaines. Elle débouche non seulement sur du code de production mais également sur un ensemble de cas de tests fonctionnels couvrant l'ensemble des mini-scénarios ou « histoires » qui la définissent. En outre, tous les composants de tests unitaires et d'intégration pertinents à l'itération auront été produits préalablement à la création du code de production, et tous devront pouvoir être exécutés sans faute avant que l'itération ne puisse être conclue.

Notons que l'itération initiale d'un cycle de développement ou de maintenance doit permettre de statuer sur les caractéristiques de l'architecture du logiciel à construire ou à modifier. Les principes de croissance du logiciel sur son squelette, préconisés par Beck (2000), sont semblables à ceux proposés par Robillard, Kruchten et d'Astous (2003) dans le cadre du UPEDU.

La phase de déploiement (*productionizing*) (Ambler 2002) constitue une extension de la phase de développement incrémental au cours de laquelle on procède aux divers tests fonctionnels sur le système et effectue les modifications découlant de ceux-ci.

1.2.3 Les processus d'ingénierie par opposition aux méthodologies agiles

La guerre de clochers que se livrent avec plus ou moins de passion les partisans des processus d'ingénierie et ceux des méthodes agiles repose sur fort peu de matériel empirique pertinent. Bien que de nombreuses études portent sur la comparaison de certains aspects précis des méthodologies agiles par rapport à leur conception traditionnelle (voir par exemple Williams, Kessler, Cunningham et Jeffries 2000, qui traite de la programmation par paires), les études comparatives entre les deux approches prises globalement (voir par exemple Noll et Atkinson 2003) sont encore rares. Une analyse appropriée du comportement réel des ingénieurs du logiciel permettrait d'éclairer grandement notre compréhension des forces et des faiblesses des méthodologies ou des processus appliqués.

1.3 La mesure des processus logiciels

La mesure des processus logiciels, considérée au sens large, fait partie des tâches reconnues par l'école classique du génie logiciel comme étant nécessaires à la production de systèmes logiciels de qualité. À titre d'exemple, le niveau 4 du Capability Maturity Model (CMM) (Paulk, Curtis, Chrissis et Weber 1993) du Software Engineering Institute est entièrement axé sur la gestion quantitative de la qualité des produits et processus logiciels. Ce niveau comprend un secteur clé intitulé « Gestion quantitative du processus », qui traite du contrôle quantitatif de la performance d'une instance de processus. L'accent est mis sur la recherche des causes spéciales de variation à l'intérieur d'un processus stable, ainsi que sur la correction de ces causes. Bien entendu, une telle démarche repose sur la capacité des intervenants à effectuer une mesure adéquate du processus. De même, le Personal Software Process (PSP) (Humphrey 1997), également du Software Engineering Institute, intègre la mesure du processus à tous les échelons de celui-ci. La mesure de processus prescrite couvre plus particulièrement la mesure de l'effort, des coûts et de la taille du logiciel, mais également celle des défauts produits, des taux d'élimination des défauts, de la productivité ainsi que de la qualité du processus au sens large.

1.3.1 L'estimation de l'effort

La mesure des processus logiciels a traditionnellement été régie par le besoin d'obtenir des estimations précises de l'effort déployé. L'évaluation des coûts d'un

projet d'ingénierie est requise afin de permettre la prise de diverses décisions ou l'évaluation de certaines informations : accord pour le démarrage ou la poursuite du projet ; construction ou achat ; détermination du prix de vente ; calcul du rendement des investissements ; allocations budgétaires (Ostwald 1992). Le génie logiciel, discipline qui constitue ni plus ni moins que l'application de l'ingénierie au logiciel (Standards Coordinating Committee of the Computer Society of the IEEE 1990), ne fait pas exception à la règle. Il faut noter toutefois que, comme les coûts de main-d'œuvre constituent généralement une très forte proportion des coûts globaux de développement et de maintenance d'un logiciel (Fenton et Pfleeger 1997; Florac et Carleton 1999), l'évaluation de l'effort requis dans le cadre d'un projet s'avère souvent le déterminant le plus important du coût prévisible de celui-ci.

Or, la qualité générale des estimations de coûts et d'échéanciers en génie logiciel est d'une faiblesse notoire (Fenton et Pfleeger 1997). Le Standish Group (2001) diffuse les statistiques suivantes portant sur l'état de l'industrie américaine du logiciel au cours de l'année 2000 :

- 28 % des projets sont terminés à temps et à l'intérieur des estimations initiales.
- 49 % des projets sont annulés avant leur achèvement ou ne sont jamais mis en œuvre.
- Les dépassements de coûts équivalent en moyenne à 45 % des estimations initiales.

– En moyenne, les logiciels livrés répondent à 67 % des besoins spécifiés.

Les statistiques laissent entrevoir non seulement une difficulté généralisée à maîtriser les paramètres qui contrôlent l'envergure d'un projet de développement, mais également un manque généralisé de contrôle du processus logiciel. C'est pourtant le processus logiciel qui permet de transformer les besoins des usagers en produits logiciels (Standards Coordinating Committee of the Computer Society of the IEEE 1990). L'incapacité de combler l'ensemble des besoins formels exprimés dans le cadre d'un projet de développement constitue clairement un échec du processus logiciel. À l'inverse, l'application adéquate d'un processus logiciel approprié doit permettre non seulement de livrer les caractéristiques fonctionnelles et non fonctionnelles prévues, mais d'y parvenir sous des conditions prévisibles de coûts et de délai de livraison et de réduire ainsi la fréquence des problèmes mentionnées précédemment.

Notons que les modèles de coût et d'effort les plus courants n'intègrent pas les caractéristiques du modèle de processus utilisé. Bien sûr, certains modèles de coûts et d'effort, comme c'est le cas du COCOMO II (Boehm, Clark, Horowitz, Westland, Madachy et Selby 1995), tiennent compte des méthodes de développement en place, de la maturité du processus logiciel ou de l'influence de facteurs purement locaux. Toutefois, l'absence de lien direct entre le modèle de coûts et le modèle de processus constitue un inconvénient notable dans le contexte de la boîte à outils du gestionnaire de projets ou de l'ingénieur de processus logiciel. Morisio (1999)

constate que, même si une attention croissante est portée à la modélisation des processus logiciels, l'établissement d'un couplage strict entre celle-ci et la mesure de processus en est à ses premiers balbutiements.

1.3.2 La mesure de l'effort

Dans le cadre d'un travail d'analyse des discussions lors de réunions d'inspection, Seaman et Basili (1998) exposent la distinction entre les méthodes de recherche qualitatives et quantitatives. Selon ces auteurs, les méthodes qualitatives permettent d'analyser des données comme des mots ou des images et sont surtout utilisées afin de générer des hypothèses, plutôt que les mettre à l'essai. Les méthodes quantitatives touchent l'analyse de données numériques et sont souvent utilisées aux fins de confirmation ou de tests d'hypothèses déjà émises.

Dans le cas de la présente étude, la nature exploratoire des travaux amène naturellement à considérer les aspects qualitatifs d'une démarche de recherche. Toutefois, l'étude, bien qu'exploratoire, a été réalisée dans le contexte de méthodologies et modèles existants et bien définis. Elle a pour thème central l'effort déployé par les développeurs, entité qui s'analyse le plus souvent de manière quantitative. Pour ces raisons, et de la même manière que Seaman et Basili, nous optons pour l'utilisation d'une approche hybride qualitative-quantitative.

Selon Robillard, Kruchten et d'Astous (2003, chap. 12), deux facteurs font de la mesure de l'effort une entreprise ardue et susceptible d'engendrer des données

difficiles à interpréter. D'abord, la mesure de l'effort nécessite un minimum d'interactions avec les individus, ce qui oblige la prise en compte de nombreux aspects sociaux et psychologiques. Ensuite, les systèmes courants de mesure de l'effort présupposent l'existence d'une relation un-à-un entre les activités mesurées et les tâches selon lesquelles l'effort est enregistré. En pratique, rien n'assure une telle adéquation. Les auteurs énumèrent quatre menaces distinctes à la validité des données d'effort, soit : la validité des mesures proprement dites, la non-uniformité des intervalles de temps, la non-uniformité de l'utilisation des jetons et l'ambiguïté des jetons. Germain, Dulipovici et Robillard (2002a) ont montré que, même en utilisant un système de mesure basé sur la classification des activités du UPEDU et sur le découpage du temps en quantas bien définis, les données recueillies montraient des signes notables d'ambiguïté et de confusion.

La littérature est pauvre en ce qui a trait aux principes de mesure de l'effort dans des contextes nécessitant la séparation détaillée de l'effort en activités. La mesure des activités cognitives effectuée par les ingénieurs du logiciel, quant à elle, est relativement récente (Robillard 1998) et a notamment été utilisée dans le contexte de l'analyse fine des échanges entre les membres d'une équipe de développement (d'Astous 1999; Seaman et Basili 1998). Cette situation peut s'expliquer en partie par le degré de difficulté associé à la mesure des processus en général. Seaman (1999) explique que les aspects d'un processus de développement de logiciels qui s'offrent à l'observation sont limités. Une partie importante du travail de développement de

logiciels s'effectue à l'intérieur de la tête des développeurs et est, par conséquent, plus difficile à observer.

1.4 Le contrôle des processus

Une des applications les plus intéressantes de l'approche préconisée au chapitre 4 touche le développement d'approches de contrôle de processus et d'estimation en cours de projet. La présente section offre un survol rapide de ces domaines. Un exposé détaillé des concepts et approches en la matière peut être trouvé dans Münch et Heidrich (2004). Bien que ces concepts ne se situent pas dans l'axe de la recherche effectuée et sont, somme toute, encore peu utilisés en milieu industriel, ils pourraient redevenir d'actualité suite aux travaux présentés dans le cadre de ce mémoire, ou dans le cadre de recherches similaires.

Un centre de contrôle de projet logiciel (Software Project Control Center ou SPCC, tel que défini par Münch et Heidrich 2004) est un outil assimilable à un système de contrôle de trafic aérien, servant de colonne vertébrale à la démarche de mesure, d'interprétation et de visualisation des métriques issues d'un processus logiciel. Le SPCC utilise un ensemble de métriques recueillies dans le cadre d'un projet en cours ainsi qu'une banque de métriques issues de projets antérieurs, et génère une vue contextualisée de ces données aux fins de leur utilisation par, notamment, les gestionnaires de projets ainsi que le personnel d'assurance-qualité. Un SPCC est formé des éléments suivants :

- un ensemble de techniques et de méthodes de contrôle de projet ;
- un ensemble de règles touchant l'utilisation des techniques et méthodes ;
- une architecture logique permettant l'interfaçage du SPCC avec son environnement ;
- un outil mettant en œuvre l'architecture logique.

Münch et Heidrich présentent également quatre techniques et méthodes pouvant être utilisées aux fins de contrôle d'un projet, soit :

- l'analyse par arbre de classification ;
- les variables dynamiques ;
- l'analyse par grappe (*Cluster analysis*) ;
- la détection des changements de tendance.

L'une de ces approches, l'analyse par grappe, peut s'avérer particulièrement utile pour l'analyse des patrons d'effort déployés au cours d'un projet. Les métriques de projet recueillies, touchant l'effort ou toute autre variable de processus, sont représentées par un vecteur multidimensionnel composé de valeurs mesurées à autant de périodes distinctes. La technique d'analyse par grappe permet de cerner des ensembles de données qui affichent le même comportement et ce, afin de répertorier des groupes de patrons similaires. Le calcul de la distance euclidienne entre les vecteurs permet la catégorisation des grappes. Li et Zelkowitz (1993) ont utilisé des vecteurs à 15 dimensions ainsi qu'un échantillon de 24 projets afin de relever les patrons émergeant de huit indicateurs distincts, dont l'effort total employé. La seg-

mentation de l'échelle temporelle a été réalisée en subdivisant chacune des quatre phases des projets en un nombre variable de portions. L'application de la technique sur les projets existants a permis d'illustrer, selon les auteurs, le pouvoir prédictif d'une telle approche. Ce constat amène naturellement à envisager l'utilisation des données empiriques recueillies en cours de projet pour la réestimation en ligne dudit projet. Le chapitre 4 décrit brièvement les travaux de MacDonell et Shepperd (2003) sur une telle approche.

CHAPITRE 2

ORGANISATION GÉNÉRALE DU TRAVAIL

Ce mémoire présente le fruit d'un travail de recherche approfondi qui a porté sur trois éditions successives du cours « Atelier de génie logiciel » (voir Robillard 1996) ainsi que sur deux approches distinctes de développement de systèmes logiciels. Les travaux réalisés ainsi que leurs conclusions sont rapportées dans les trois chapitres qui suivent.

Le chapitre 3 introduit le premier des deux articles qui constituent le corps du présent travail. L'article, intitulé « Engineering-Based Processes and Agile Methodologies for Software Development : A Comparative Case Study », par Éric Germain et Pierre N. Robillard, a été accepté pour publication par la revue *The Journal of Systems and Software*. L'étude de cas présentée couvre l'application de deux approches distinctes de développement de logiciels dans le cadre de l'implantation de versions multiples d'une même spécification de requis logiciels. Elle a nécessité la définition d'une catégorisation originale des activités cognitives réalisées par les sujets de l'étude, qui étaient des étudiants inscrits à l'édition 2002 de l'Atelier. Trois des équipes ont utilisé un modèle de processus très fortement inspiré du *Unified Process for Education* (UPEDU), modèle de processus dérivé du *Rational Unified Process* (RUP). Les trois autres équipes ont utilisé un processus construit

autour de la méthodologie de programmation extrême (*Extreme Programming* ou « XP »).

Le chapitre 4 constitue le second des deux articles qui forment le corps de ce travail. L'article, intitulé « Measuring Relative Effort Progression Paths in the Discipline Space of the Software Engineering Process », également par Éric Germain et Pierre N. Robillard, a été soumis pour publication à la revue *IEEE Transactions on Software Engineering*. L'analyse présentée vise à cerner des patrons au sein des diagrammes de répartition de l'effort relatif en disciplines à travers le cycle complet de développement d'un système logiciel. Huit équipes d'étudiants, issues de trois éditions différentes de l'Atelier (2001, 2002 et 2003), ont utilisé un modèle de processus très fortement inspiré du UPEDU afin de développer des systèmes logiciels distincts. Les spécifications soumises, les durées de cycle de développement, les technologies utilisées et les mécanismes de cueillette de l'effort déployé variaient à divers degrés d'une édition à l'autre. Le nombre de participants différait également d'une équipe à l'autre. Une catégorisation en trois disciplines des activités réalisées a été définie de manière à permettre une ségrégation substantielle des activités tout en minimisant l'influence des facteurs liés à l'édition ou à la taille de l'équipe. L'article comporte une annexe, dont le texte est présenté à l'annexe I du mémoire.

Le choix et l'ordonnancement des articles intégrés dans ce mémoire sont justifiés par la nécessité de clarifier successivement les diverses sources d'influence pouvant s'exercer sur l'effort déployé. Avant la réalisation du présent travail de recherche, les

limites de la valeur potentielle de l'Atelier comme plate-forme de recherche n'étaient pas clairement établies. L'Atelier de génie logiciel constitue certes une plate-forme très intéressante pour l'étude du comportement des développeurs ; toutefois, un inconvénient notable du concept est le faible nombre d'équipes participantes lorsque chaque édition est considérée séparément. Cet inconvénient empêche l'utilisation des outils statistiques conventionnels aux fins d'une analyse formelle de la convergence des comportements d'une équipe à l'autre. Il importait donc de clarifier les limites réelles découlant de ces contraintes.

Le premier article permettait d'identifier l'influence du modèle de processus sur l'effort déployé par les participants au niveau d'agrégation choisi. Une influence indue du modèle de processus sur le comportement cognitif des développeurs aurait grandement limité la capacité de généralisation de l'analyse au milieu industriel. Par ailleurs, l'étude visait également à établir le degré approprié de granularité des disciplines utilisées aux fins de l'analyse de la répartition de l'effort. En ce sens, le premier article a permis d'ouvrir la voie à l'approfondissement de la question des patrons d'effort dans la perspective du degré d'avancement d'un projet.

Quant au second article, il vise à ouvrir une fenêtre permettant de sortir du carcan de l'analyse en mode mono-édition, en montrant que certains patrons d'effort transcendent les caractéristiques de projet et d'équipe. Cette démarche est cruciale pour l'application éventuelle des analyses de patrons vers le milieu industriel, où la présence de projets similaires réalisés en parallèle par des équipes de développement

distinctes constitue l'exception plutôt que la règle.

En somme, le choix et la séquence des articles intégrés dans ce mémoire reflètent une progression séquentielle naturelle des travaux vers la réalisation future d'analyses de patrons d'effort en milieu industriel.

Par ailleurs, trois articles de conférence auxquels l'auteur a contribué ont été inclus sur un disque compact qui accompagne ce mémoire. Le contenu du disque est présenté brièvement à l'annexe II. Les deux premiers articles (Germain, Dulipovici et Robillard 2002a; Germain, Robillard et Dulipovici 2002b) portent sur l'édition 2001 de l'Atelier et illustrent de façon préliminaire la faisabilité de l'approche entreprise dans la présente recherche. Ils illustrent également les difficultés inhérentes à la démarche, notamment en soulevant la question de l'ambiguïté et de la confusion de la classification en activités de processus. Le troisième article (Germain et Robillard 2003) constitue une version préliminaire du chapitre 3 du présent mémoire.

CHAPITRE 3

ENGINEERING-BASED PROCESSES AND AGILE METHODOLOGIES FOR SOFTWARE DEVELOPMENT : A COMPARATIVE CASE STUDY

Abstract

The emergence of various software development methodologies raises the need to evaluate and compare their efficiencies. One way of performing such a comparison is to have different teams apply different process models in the implementation of multiple versions of common specifications. This study defines a new cognitive activity classification scheme which has been used to record the effort expended by six student teams producing parallel implementations of the same software requirements specifications. Three of the teams used a process based on the Unified Process for Education (UPEDU), a teaching-oriented process derived from the Rational Unified Process. The other three teams used a process built around the principles of the Extreme Programming (XP) methodology. Important variations in effort at the cognitive activity level between teams shows that the classification could scarcely be used without categorization at a higher level. However, the relative importance of a category of activities aimed at defining "active" behaviour was shown to be almost constant for all teams involved, possibly showing a fundamen-

tal behaviour pattern. As secondary observations, aggregate variations by process model tend to be small and limited to a few activities, and coding-related activities dominate the effort distribution for all the teams.

3.1 Introduction

There is increasing interest in the teaching of software processes (see, for instance, El Emam 2001; Halling, Zuser, Köhle and Biffl 2002; Höst 2002; Robillard, Kruchten and d'Astous 2003; Umphress and Hamilton 2003). However, this interest has not yet translated into acceptance of a common set of process principles. Rather, two main software process approaches seem to be emerging. The first promotes the utilisation of a discipline-based engineering process involving effective definition of the roles to be played, activities to be performed and artifacts to be produced. This approach involves the production of artifacts to support early decision-making on requirements and design matters, effective communication, knowledge reuse and mutual work inspection. The main principle here is that efforts made in up-front planning activities and in artifact production will result in lower overall cost, timely product delivery and better software quality. The Rational Unified Process (RUP) (Kruchten 2000) is an example of a process that fits this approach. The Unified Process for Education (UPEDU) is a RUP-based process designed for teaching the software processes in software engineering and computer science programs. The process is built on a set of focused activities with significant

cognitive content, and on a set of essential artifacts that should be needed for small projects. The process also defines basic roles such as "Designer", "Implementer", "Tester", "Project Manager" and "Reviewer" which are easily understandable by developers who are inexperienced. Roles, activities and artifacts are grouped into disciplines, which are categorized as either engineering or management. The UP-EDU has been extensively described in Robillard, Kruchten and d'Astous (2001, 2003) and is readily available online (École Polytechnique de Montréal 2002).

The other approach, called "Agile Software Development" (Agile Alliance 2003; Cockburn 2002), promotes quick response to changes in requirements as well as extensive and ongoing collaboration between the development team and the customer. The approach specifically downplays the importance of formal processes and comprehensive documentation (Agile Alliance 2003). It is based on the assumption that one cannot truly anticipate project requirements right at the beginning of a software development project, and that the proper way to deliver timely, quality software in a cost-effective manner is instead to build flexibility within the development activities. The Manifesto for Agile Software Development (Agile Alliance 2003) provides the basic values for agile development in detail. Some methodologies inspired by this approach include Adaptive Software Development (Highsmith 2000), Scrum (Rising and Janoff 2000), the Crystal family (Cockburn 2002; Highsmith 2002), Feature-Driven Development (Palmer and Felsing 2002), Dynamic System Development Method (Stapleton 1997) and Extreme Programming (XP;

see Beck (1999)).

There is no such thing as a universal process or methodology. Organisations will typically develop their own process or methodology based on the approaches described, and often on an existing process model. For example, an organization may use tools provided with the RUP package to build a custom process containing a chosen subset of the proposed activities and artifacts. Alternatively, an organization may simply take the generally recognized set of XP practices and adapt them to its particular context.

Meanwhile, students enrolled in a software engineering program who have received training on software processes are expected to be, at the end of the program, more sensitive to issues affecting software quality, cost and life cycle. This does not mean, however, that those individuals will actually apply everything they have learned. Previous studies (Germain, Dulipovici and Robillard 2002a; Germain, Robillard and Dulipovici 2002b; Robillard 1998) in the context of the Software Engineering Studio, a project-oriented course for senior-level students, have shown a significant gap between theory as taught and practice. In these studies, "effort slips" were used as an indicator of relative activity intensity. Analyses performed were, however, limited by the activity and artifact classification of the UPEDU-based process used, making it rather difficult to determine exactly which elemental tasks had been performed.

Using those studies as a foundation, we defined a set of cognitive activities aimed

at recording the various activity states of a software developer during the course of a project. The utilisation of such a classification allows us to study the impact of software process notions learned on the cognitive activities actually performed by the students.

An important benefit of such an approach is that it opens the door to comparative studies of the two approaches. Advocates of these are keen to debate the respective merits of each, or even sometimes the fact that there is any dichotomy at all between them (see, for instance, Antón 2003; Beck and Boehm 2003; Wells 2003). Several studies have been performed to analyse the value of a particular practice (see, for instance, Cockburn and Williams 2001). However, very few studies have compared two complete software process approaches. This paper presents a case study of the implementation of our cognitive activity classification to the comparative analysis of two process models, one based on the UPEDU, the other based on XP.

These observations should not be readily generalized to industrial practices because of the academic nature of the setting and the impact of the particular project, life cycle and technology chosen. Besides, repeating such an experiment in an industrial setting would be quite difficult because of the need to record individual cognitive activities at the developer level. However, we think that the study presented in this paper provides clues which may be useful when evaluating the importance of a specific software engineering process.

3.2 The Software Engineering Studio

The Software Engineering Studio is an optional project-oriented course offered to senior-year students in computer engineering at the École Polytechnique de Montréal. Its purpose is to allow students to gain practical experience in software development by participating in a small-scale software development project. Teams of students must develop a complete implementation based on software requirements specifications provided by the instructors. Also, they must use a well-defined software engineering process. Participants thus acquire experience early on in building an entire operational software project through design, implementation, testing and management activities. This project course teaches them the realities of teamwork, milestones and time constraints. As a secondary objective, students become more familiar with a specific application domain or set of technologies. An earlier version of the Studio is presented in detail in Robillard (1996). The Studio has also served as a test bed for the study of development effort and artifact quality. Some individual studies performed using data generated in a Studio have been documented in Germain, Dulipovici and Robillard (2002a), Germain, Robillard and Dulipovici (2002b) and Robillard (1998). Several other software engineering programs implement other paradigms of the concept of a Studio. For instance, Blake and Cornett's Software Engineering II class (Blake and Cornett 2002) implemented a setting that was aimed at teaching every step of a typical software development cycle to students split into specialised teams in the context of a single, class-wide project. By

contrast, in our approach, students work in small teams, each developing their own version from common specifications. Also, in our setting, students are free to organize their work however they wish, as long as they deliver the required artifacts at each defined milestone.

The Winter 2002 Studio featured the development of a Web-based meeting management system aimed at organisers of meetings where the number and geographic dispersion of participants make scheduling difficult. The software system to be developed would allow meeting coordinators to send availability requests to a set of individuals, so that each of them can specify personal availability periods. The set of availability periods would then be graphically represented using a special calendar tool to enable a coordinator to visualise the relevant information at a glance, making the scheduling decisions easier. The decisions can then be transmitted electronically to all participants. The software system would be responsible, among other things, for ensuring proper data storage and updates, as well as for communications among all participants. All communications would be performed using standard e-mail. All the students had to use identical software requirements specifications and were instructed not to add any additional features not explicitly requested. Figure 3.1 shows a screenshot from one of the software products delivered.

The main feature of the Winter 2002 Studio was the use of two different software engineering processes. Although all the teams were competing on the same project,

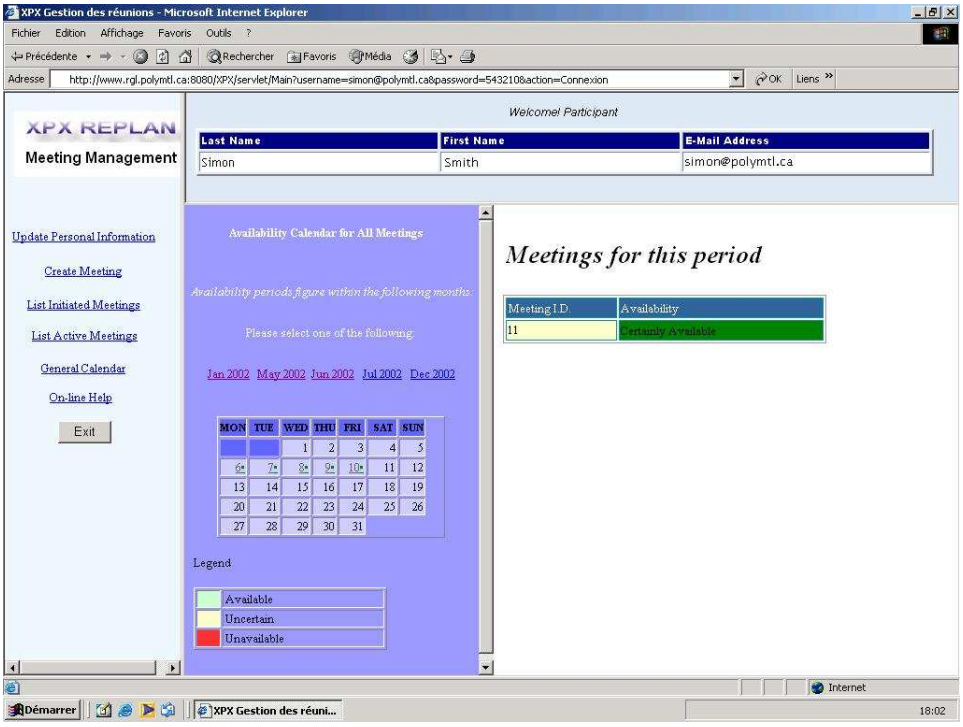

Figure 3.1: Screenshot from a completed product

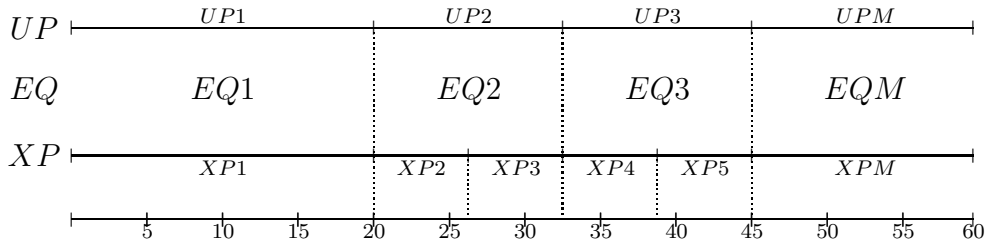

Figure 3.2: Software development schedule (in working days)

each team would follow one of the two proposed software development approaches. One of the goals of the study was to determine the influence of the software process on the participants' behaviour. Thus, three of the teams were assigned a process based on the UPEDU. The other three teams were assigned a process built around the Extreme Programming methodology.

Figure 3.2 illustrates the prescribed software development schedule. The top row shows the timeline of iterations for UPEDU, while the bottom row shows the same information for XP. The lower scale represents the nominal number of academic calendar days since the beginning of the project. For example, iterations UP3 and XP5 both finished on the 45th day of the project. The middle row illustrates the timing of the equivalent iterations that had been defined for data analysis purposes and which are explained below.

A common release-level framework was used to define the development schedule

for both processes. Thus, for all the teams, an initial specifications document was provided at the beginning of the semester. A complete implementation of that specification was due after 45 days. Thereafter, a second specifications document was issued which requested modest architectural changes to the system. Implementation of those changes was due following a further 15-day period. Iterations XP1 through XP5 and UP1 through UP3 belong to the initial development cycle, while iterations XPM and UPM belong to the maintenance phase. An important element to mention is that no specific constraints were imposed for specific interations, other than the requirement to deliver specification-compliant end-products and linked artifacts at the end of the development cycle. In other words, all activities could be performed at various levels within all iterations. Thus, teams had complete freedom to schedule development work throughout the development cycle. Evaluating workload and risks as well as scheduling development work are part of the tasks that a software engineer must be able to perform on an autonomous basis.

The concept of iteration applied here complies with the definition found in Robillard, Kruchten and d'Astous (2003):

> An iteration is a mechanism that enables the interaction between the software life cycle and the software process by defining milestones that are often associated with the phases of the product. Sets of iterations are also used to define the software process.

Iterations are common to both traditional and agile processes. However, the average size of a single iteration is likely to vary depending on the process model used. Beck (2000, Glossary) defines an iteration as "a one- to four-week period", while the UPEDU (École Polytechnique de Montréal 2002, "Guidelines: Software Development Plan") considers iterations of less than one month as short and "... more suitable for the Construction phase, where the degree of new functionality to be included and the degree of novelty are low." One reason for these differences is simply that people using the UPEDU require some considerable time to produce a significant amount of documentation, while people using XP are likely to choose not to do so. At the same time, one may argue that the additional investment in artifacts should enhance the productivity of the software engineers using UPEDU. However, this claim has not been proven and rightly constitutes one of the aspects that was studied in this setting.

We chose, therefore, to arbitrarily customise iteration lengths for each of the processes used. Thus, the development schedule for the XP process would include a larger number of iterations covering the same time frame. However, the initial iterations time frame at the beginning of the project was of particular concern to us. The first iteration is crucial for laying out the skeleton of the system. Also, the instructors wanted to leave enough time for every participant to become accustomed to the development environment provided and minimally comfortable with the language and technologies. The initial iteration has therefore been kept

identical in length for both processes. Also, the maintenance cycle has been limited to a single iteration following general agreement by the students that this would be sufficient considering the limited scope of the changes requested. The remaining iterations were set out such that the ratio of XP iterations to UPEDU iterations would be 2:1. Correspondence of UPEDU iteration end-dates to XP iteration end-dates was required to facilitate analysis of the resulting data in terms of effort expended. A set of equivalent iterations, EQ1, EQ2, EQ3 and EQM, was defined for that purpose.

Students were free to team up with whomever they wished to and to register for the process model they wanted to use, provided that the following three conditions were fulfilled: We had to have five teams of four students each and one team of three students. Three of the teams had to use the UPEDU, and the other three had to use the XP-based process. The three-student team had to use the UPEDU, since three people cannot all be pair-programming at the same time. The only problem we ran into was that two students were competing for the fourth seat on one UPEDU team. We chose to break the tie by drawing a winner's name.

There was no preliminary assessment of the participants' skills, as it was assumed that all the students were equally capable of performing their tasks correctly and efficiently. However, some variations did exist among individuals. A small number of students were not yet familiarized with the UPEDU. Also, a small number of students had been exposed to the Java language and to related technologies. Nev-

ertheless, we feel that these differences did not create any significant asymmetry among the teams.

3.3 Cognitive activity classification

In previous Studios, students had been asked to record effort expended under each process activity. That approach has the benefit of allowing a direct measurement of the process itself. However it can only be used on the assumption that the list of activities defined in the process covers every possible work situation without generating excessive overlap.

Such an assumption has not been confirmed. Indeed, an analysis performed using data from the 2001 Studio (Germain, Dulipovici and Robillard 2002a) showed the possible presence of ambiguity and confusion among participants in relation to the process activities as defined by the instructors. Another problem with the approach was that the presence of two separate software processes prevented the utilisation of a single process-based scheme that would allow comparison of the effort expended by all the teams. An alternative approach was to use a process-independent classification which would lead to the implicit assignment of effort to the proper activity. A classification based on the evaluation of explicit, mutually exclusive cognitive activities constituted an interesting path to this target.

Table 3.1 illustrates the classification that was used for the the study. Participants were presented a total of 14 cognitive activities and asked to tag each effort

Table 3.1: Cognitive activity classification

	Preparation	*Implementation*	*Control*
Active	Draw Write	Code Code & Test Integrate & Test Test	
Reflexive	Browse / Search Read Think		
Interactive	Discuss		Inspect / Review

slip to be recorded with one and only one of these activities. Three of these activities dealt mostly with tasks that we wished to exclude from our study, such as training and systems administration. These activities are not included in Table 3.1, and all effort slips recorded under them have been excluded from this study.

Although most activity names are self-explanatory, we provide a short description of them below. Two distinct groupings of cognitive activities were defined for analysis purposes. These groupings were not revealed to the participants. Each represents one dimension in Table 3.1. The grouping along the horizontal axis, called "step-oriented", represents the sequence of steps typically performed in the development of a software product. The grouping along the vertical axis, called "state-oriented", illustrates the cognitive states likely to be taken during software development.

The "Active" state occurs when a developer is performing any activity the purpose of which is to generate or modify an artifact. It includes the cognitive activities "Draw" and "Write", which refer to the production of diagrams and text of all kinds

respectively. It also includes a set of activities which collectively cover everything related to coding, integration and testing: “Code”, “Code & Test”, “Integrate & Test” and “Test”. This redundant labelling of activities was necessary because, under XP, these activities are often intertwined in practice. Thus, all plausible combinations of coding and testing are taken into account. Also, the “Integrate & Test” activity reflects the fact that integration is presumably a short duration activity which immediately leads to testing.

The “Reflexive” state occurs when a stand-alone developer is performing any activity which does not generate or modify any artifact. The “Browse / Search” activity refers to the action of reading documents or Web pages in a non-specific order, as when searching for documents that will eventually be read, while the “Read” activity refers to the action of reading a specific document such as a textbook or article for the purpose of assimilating a chunk of information. The “Think” activity refers to the process of self-reflection and thus encompasses every effort expended by a stand-alone participant for which no input or output exists.

The “Interactive” state refers to work performed by two or more teammates at the same time. The “Discuss” activity refers to every discussion taking place between a team member and one or more individuals who may or may not be team members. The “Inspect / Review” activity refers to the technical review activities that may be performed following the production of any artifact.

The “Preparation” step refers to activities which may be considered as prereq-

uisites for coding, while the "Implementation" step covers all coding, integration and testing effort. The "Control" step corresponds to explicit quality control work and includes the sole activity "Inspect / Review".

The original classification included other activities the occurrence of which would be interpreted as merely accidental and weakly linked to the fundamental behavioural characteristics of the participants. We removed all slips declaring any of those activities from our subsequent analysis.

3.4 Analysis of cognitive activities performed

3.4.1 Raw-data analysis

The first analysis step is to look at the raw data for each team based on the effort slips that were submitted. Figure 3.3 illustrates the effort slips corresponding to the cognitive activities of each XP team (XPA, XPB, XPC), and of each UPEDU team (UPA, UPB, UPC). The state- and step-oriented categories are shown along the left and right sides of the diagram respectively. We chose at this point to remove all data related to the maintenance phase, which was initially set up to measure the relative maintainability of each software system. We suspect that the various design choices made by the individual teams may have had a significant impact on the effort patterns deployed, and therefore on their convergence.

Potential sources of team productivity variations are many; for instance, productivity gaps between individual developers are often very important (see, for

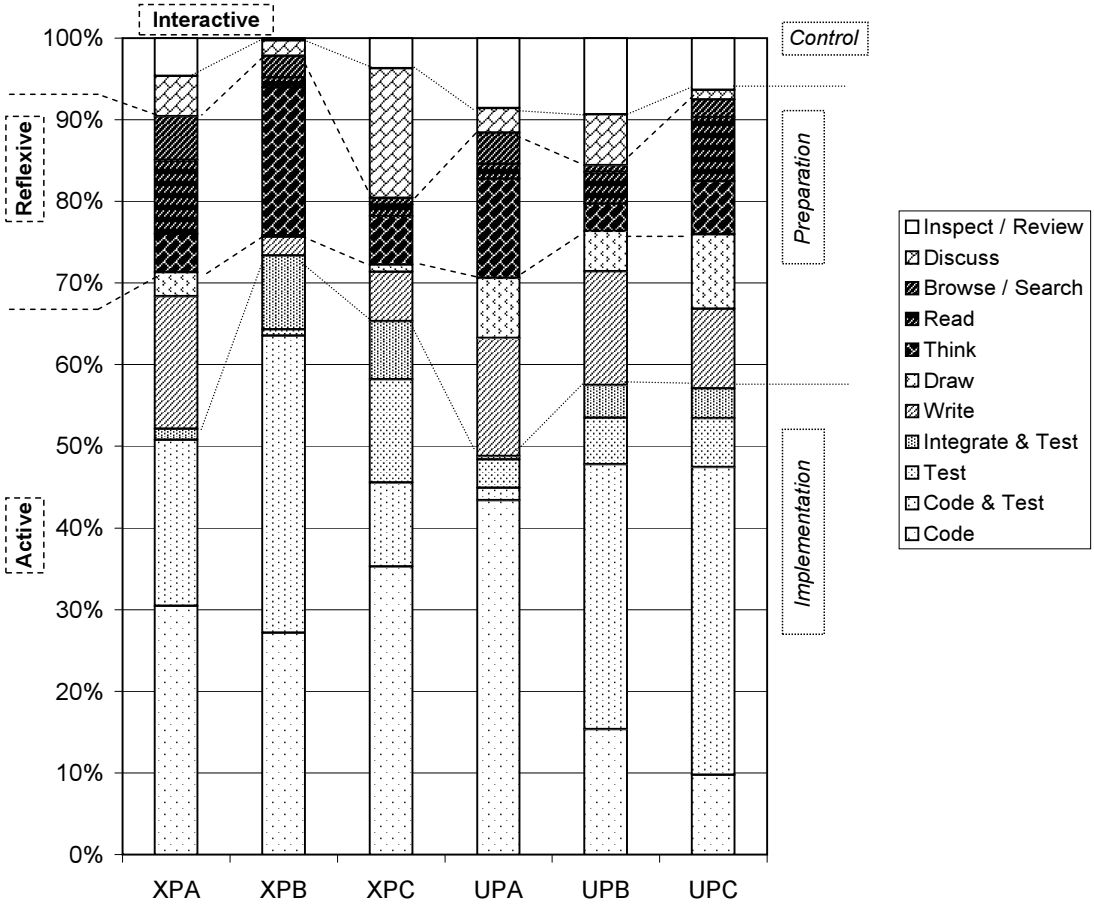

Figure 3.3: Effort distribution by activity, by team

instance, Brooks 1995, chapter 3) and may generate even wider gaps at the team level (Cockburn 2002, chapter 4). In order to eliminate the influence of such variations, all effort data has been normalized relative to total effort for a given team.

Figure 3.3 clearly shows wide variations in intensity among teams for each cognitive activity taken individually. This result shows that effort convergence at the cognitive activity level is low. However, relative effort expended on "active" cognitive activities (see left side of Figure 3.3) falls into the 71% to 76% range, which is narrow. This suggests a very strong inherent behaviour characteristic which is independent of the process model used. By contrast, effort distribution between the "Reflexive" and "Interactive" categories varies widely from team to team, and likewise shows no relation to the process model.

Implementation activities (see right side of Figure 3.3) represent between 49% and 73% of total effort for each team. This clearly shows that, beyond the central analysis, design and testing skills that are, rightly, promoted within the software engineering community, this discipline remains a coding-intensive one, even when performed by students aware of the importance of software process activities. This finding might provide a part of the answer to the question raised in McConnell (2002): "How important is software construction?" Software construction is very important indeed, at least in terms of its intensity relative to other categories of activities.

The only category showing any apparent influence of the process model is the

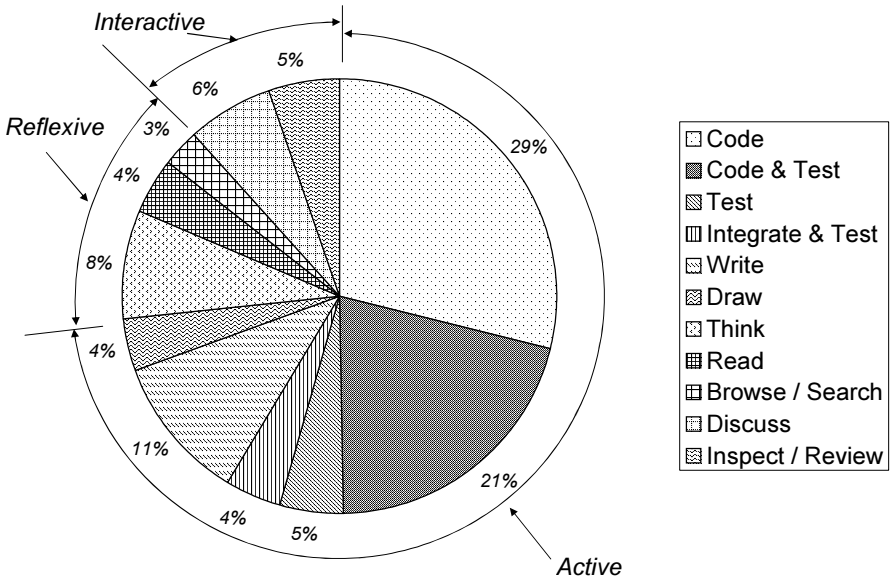

Figure 3.4: Aggregate effort distribution by cognitive activity

"Control" category, where all UPEDU teams expended more effort (average of 8%) than XP teams (average of 3%).

3.4.2 Aggregate analysis

Figure 3.4 illustrates the aggregate effort distribution by activity as a percentage of total effort expended for all teams. The diagram also shows the relative amounts of effort for each state-oriented category. The "Code" and "Code & Test" activities have aggregate relative weights of 29% and 21% respectively. All other activities have weights in the 3% to 11% range. Activities in the "Active" category include almost 75% of total effort.

Figure 3.5 shows the aggregate information for each process model. The sep-

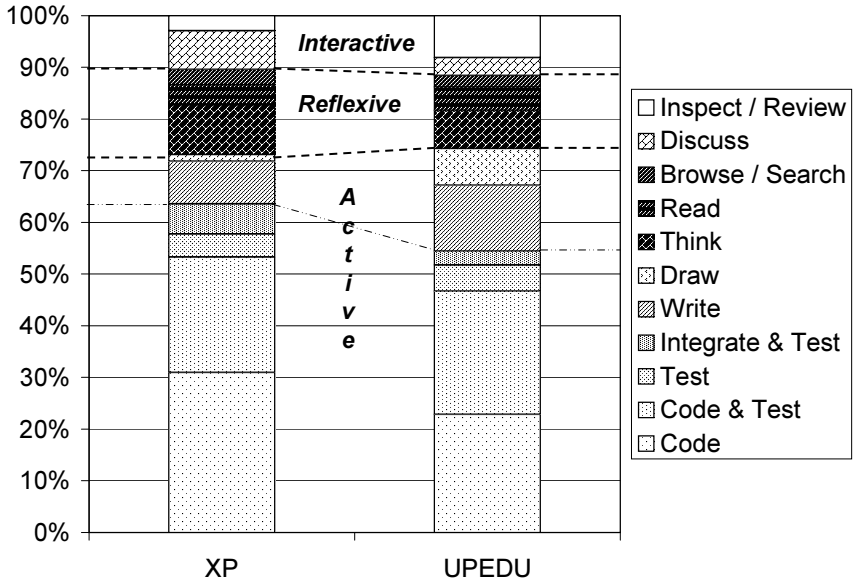

Figure 3.5: Aggregate effort by cognitive activity, by process (%)

aration between state-oriented categories is shown with bold dotted lines, while the separation between the "Draw" and "Write" activities and the other activities in the "Active" category is shown with thinner dotted lines. Coding and testing activities are less intensive under the UPEDU. However, the "Draw" and "Write" activities include more effort under this process model, bringing total effort under the "Active" category to almost identical levels for the two process models.

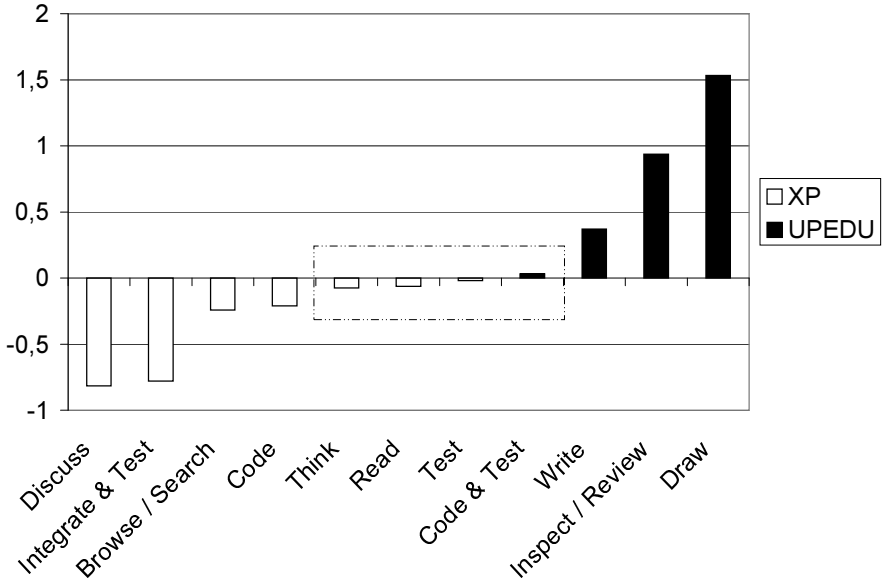

Figure 3.6: Weight index by cognitive activity

3.4.3 Inter-process analysis

Figure 3.6 illustrates the relative weight of each cognitive activity in the two process models. The *Process Activity Index* (PAI) for a given activity is defined as

$$PAI = \ln(E_{UP}/E_{XP})$$

where E_{UP} and E_{XP} represent the amount of relative effort performed by the UPEDU and XP teams respectively for that activity. Thus, an index value of zero indicates identical effort for a given activity. Negative index values mean that the effort is greater for the XP teams, while positive index values mean a greater effort for the UPEDU teams. Activities are ordered by increasing index value.

Figure 3.6 illustrates the essence of the cognitive activities of the two processes. Positive indices shown for the "Draw" and "Inspect / Review" activities were to be expected, since those cognitive activities, while required under a UPEDU-like process, are not covered at all by the XP methodology. Under XP, production of diagrams is not recommended by some of the principal gurus (see, for instance, Jeffries, Anderson and Hendrickson 2001, Ch. 16), while inspection and revision are theoretically performed "on-line" in the course of regular pair-programming tasks. At the other end of the axis, the explicit role emphasis on the "Integrate & Test" activity reflects the intertwined nature of integration and testing under XP. That the activity is performed mainly by XP-based teams should not be considered a surprise. The presence of the "Discuss" activity at the very left end of the axis posed more of a question, since the XP methodology promotes several teamwork practices such as pair programming. It is interesting to observe that each activity in the "Interactive" category is dominant in one of the process models. This suggests a real influence of the process model on the relative intensity of these activities. The framed activities in the middle of Figure 3.6 are considered to be equivalent for the two process models, since their index values are close to zero.

Figure 3.7 illustrates calculation of the *PAI* using the state- and step-oriented categories. The "Control" category, which encompasses little total effort, is overexposed here because the index calculation is based on a simple logarithmic function. The other five categories are considered to be equivalent for the two process models.

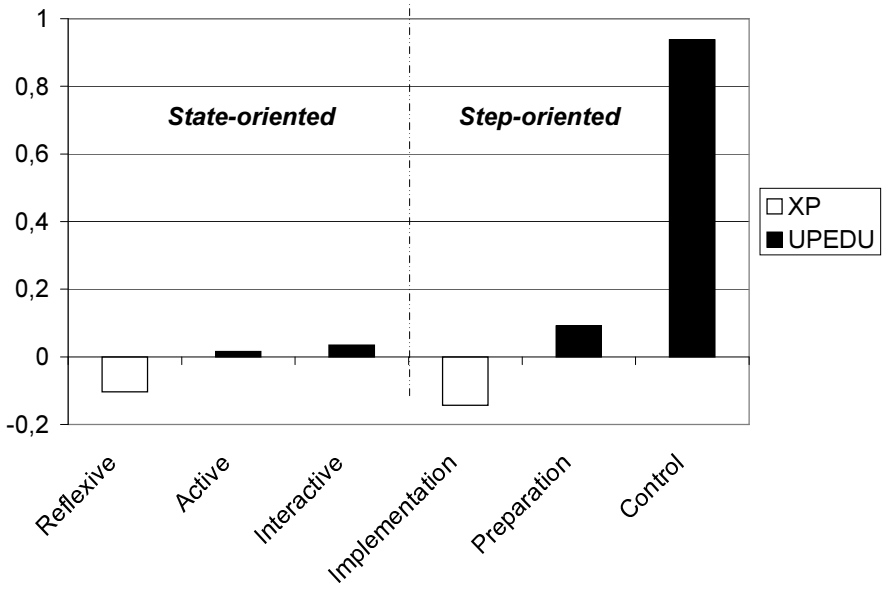

Figure 3.7: Weight index by category

3.5 Concluding remarks

3.5.1 Study limitations

Developing software is an open problem. There can be as many solutions as there are individuals or teams. In this study, all the teams provided acceptable software products in relation to the requirements specifications issued. All were also constrained by a development schedule and met all deadlines.

Even though the software process used seemed to have an impact on the importance of some cognitive activities, we did not observe any significant relation between the process used and the magnitude of the overall effort. Effort expended by the XP teams as a whole indeed exceeded effort expended by the UPEDU

teams by a whopping 29%. However, the external factors that may have affected this figure are numerous and make this result highly questionable. The only three-participant team was a UPEDU team, which showed the least total effort of the six. Also, the project required quick learning of the Java Servlet technology by the participants. Since the XP teams had to start coding almost immediately, they faced technological difficulties earlier than the UPEDU teams. We observed significant technology-related knowledge transfers from the XP teams to the UPEDU teams at the time when the latter started producing code. It must be noted that other kinds of knowledge transfers, for instance those related to architectural decisions, seem not to have occurred. Traces of such transfers have not been found in the resulting artifacts, except in the form of reuse of a few key external components by more than one team. Total absence of knowledge transfer would have been very difficult to achieve in practice.

Effort performed in the context of this analysis made up the entire contribution to the realization of a frozen software requirements specifications document. Using evolving requirements in the course of a project would bring a very interesting dimension to such a study in the future. Requirements evolution is a usual part of software development and constitutes a strong basis for the justification of the XP approach by its promoters. However, we predict that the impact of such a change on the effort mix would be quite limited unless some other major changes are implemented in the setting, such as making teams swap their respective projects after

a delivery milestone. In any case, it is important to understand that responsiveness to change is highly influenced by the previous design choices made by a team, which in any case suffers from lack of experience. Measuring effort expended in such a context would therefore include the additional challenge of trying to exclude the influence of that factor, which may prove very difficult in practice.

Two important objectives are at the centre of the case for engineering-oriented processes. First, well-defined design activities and artifacts are meant to ensure that important design decisions, especially at the architectural level, are made properly and before the costs of change become too high, assuming these costs will rise exponentially as generally assumed in the "traditional" software engineering community (Pressman 2000). Second, the value of the various artifacts that are developed through the project will be felt the most when another development team tries to maintain the software product, particularly if the original team members are not available for consultation. Unfortunately, our study neither confirms nor denies those statements. The benefits of mandatory design activities and artifacts are most evident in large projects, while the setting featured a small project. Measuring those benefits would require the realization of a similar study using a much larger project. Also, the impact of the methodology on product maintainability could only be measured through the realization of maintenance activities on software products developed using different methodologies. The Studio class as established can constitute a very suitable environment for such maintenance projects.

The validity of the study rests heavily on the level of confidence that exists in the effort slips entered by the participants. Difficult experiences from earlier editions of the Studio show that the instructors cannot wait until the end of the semester to validate the data as recorded. Such validation must occur in a continuous manner and be enforced through weekly meetings with participants. All irregular situations, for instance dubious effort slips or time periods without any entry, were signalled to the students as soon as detected, and immediate corrective action was requested in all cases. This attitude has generated two benefits. First, it helped improve the effective quality of the recorded data. Second, it gave the students the impression that the instructors really cared about effort slip quality and were determined to ensure it. This pushed the students to act proactively and prevent problems by paying careful attention to their task of accurately recording the effort expended.

3.5.2 Observations on the methodology

The main goal of this paper was to define a methodology that would enable comparison of effort performed by developers at the cognitive level, so that fundamental behaviour characteristics of a development team which are independent of the process used can be identified. In this respect, the consistent balance of effort intensity between activities in the "Active" category and other activities in the state-oriented categorisation seems to reflect such characteristics. Such an observation opens the door to the development of an effort prediction paradigm which

would use compound program size estimation techniques encompassing a whole set of artifacts, instead of just code length, for instance. Research activities prerequisite to this goal include validating this supposed invariance of the "Active" category, and measuring the result of restructuring the underlying activity classification. Ultimately, a useful target would be to define a cognitive activity set that is powerful enough to provide invariance at a very fine granularity level. This is far from being achieved as yet, since variability between teams at the individual cognitive activity level is very considerable. In the meantime, there might be some important benefits to testing the hypothesis of invariance of the "Active" category in a few industrial settings under various processes or methodologies, in order to establish its validity and usability in practice.

We applied the principle of effort normalization relative to total effort by teams throughout this study, and thus did not analyse in detail the relation between overall effort and its distribution by cognitive activity. Such an analysis would be useful, but would require a better understanding of the factors that influence total effort output for a given team, such as team size and schedule compression. This, in turn, would be best achieved using a larger number of teams, as well as multiple studio editions. Such a strategy would allow the identification of factors of influence coming from particular conditions.

3.5.3 Participants' behaviour

This study illustrates, among other things, a basic observation of team software development based on two different software engineering processes. In spite of the limited scope of the study, a few general conclusions can be drawn, although these conclusions need more experimentation in order to be validated.

The effort expended on core activities within each development project is more or less independent of the software engineering process used. Rather, one process will emphasise a particular type of activity over another. This shifted emphasis does not have a spectacular effect on the overall distribution of the cognitive activities performed. One possible interpretation is that some cognitive activities will require a minimal effort investment regardless of the software process used.

We observe that a well-defined software process such as the UPEDU will put more emphasis on the engineering aspects of the software implementation by stressing the pre-coding activities, while the XP-based process will put more emphasis on ad hoc communications. While these observations are totally in line with the definition of the processes involved, what is most interesting is that these differences between processes are simply not as great as one may have expected and have no impact on the effort-intensive coding-related activities.

We did not observe any significant relationship between the requirement for explicit artifacts for the UPEDU teams and the productivity of those teams. Informal discussions with the participants who used the UPEDU revealed that most of

them did not feel the benefit of having produced those artifacts. One possible explanation is that, since the project was very small, most relevant information could be mastered easily by the participants and communicated verbally, with greater efficiency than through the time-consuming production of documents. Of course, it is possible that such an explanation may represent more of an impression than a concrete reflection of the true characteristics of developer behaviour.

3.5.4 Teaching software development

The main advantage of a well-defined software engineering process is that students are aware of the various activities that are likely to be needed for the proper development of a software product. They may thus decide to put more emphasis on a particular activity while at the same time being able to anticipate the impacts of their decisions. It is not clear whether XP as a whole is suitable for teaching at the undergraduate level or not. It is already recognized that XP is aimed at teams of talented and experienced individuals. Indeed, experience often comes from using a more structured approach, from which one can pick the valuable aspects and throw away the rest. XP is often learned through the pairing of an experienced programmer with a junior one. Academic environments, at least in their usual form, do not fit in well with such an approach. However, it may be important to familiarise students with the nature of XP (or another agile methodology) once they have been trained in a more classical process approach. Of course, some methods which have

emerged in the light of XP, such as pair programming (see, for instance, Williams and Upchurch 2001), can be considered for teaching purposes independently of the process involved.

Acknowledgments

We are grateful to Mihaela Dulipovici who participated in the preparation of the project course, acted as teaching assistant for the course and was deeply involved in artifact and effort slip quality evaluation. Also, this project would not have been possible without the participation of all the students enrolled in the Software Engineering Studio course during the Winter 2002 semester. We would also like to thank Alexandre Moïse and Martin Robillard for their insightful comments while we were building the requirements specifications for the semester project.

This work was partly supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under grant A0141.

CHAPITRE 4

MEASURING RELATIVE EFFORT PROGRESSION PATHS IN THE DISCIPLINE SPACE OF THE SOFTWARE ENGINEERING PROCESS

Abstract

This paper presents the analysis of relevant effort patterns in the work performed in the context of a capstone project course in software engineering at the École Polytechnique de Montréal. Eight teams of students from three annual editions of the Software Engineering Studio were given the task of providing a detailed record, by activity, of their work, while each team was engaged in developing a complete and acceptable software system. The study defines a new categorization of process activities in three disciplines. The analysis shows that we can define relative effort paths in the discipline space. These paths have well-defined characteristics which may be used to monitor project progression.

4.1 Introduction

Model-based effort estimation rests on the principle that total software project effort should be an approximate function of a set of predictors. Lines of code, function points, programming language, rate of reuse, requirements volatility, software

change rates, product complexity, database sizes, non-functional requirements and platform difficulty are representative of the kinds of predictors found in models such as those of the COCOMO family (Boehm, Clark, Horowitz, Westland, Madachy and Selby 1995). These predictors refer to product characteristics or, sometimes, to key factors of process capability such as personnel experience and schedule tightness. These approaches assume that the software process generates similar outputs under similar quality and cost conditions when provided with similar inputs. Although that view may be correct for organizations in control of their software processes, it should not be assumed to be so for others. Software processes need to be monitored and controlled just like any manufacturing and business processes. Failure to follow the prescribed activities may lead to the execution of a quite different process from the prescribed one. Florac and Carleton (1999, p. 9) mention that software processes can perform poorly, or be unstable, non-compliant or incapable of delivering products that meet requirements. Unstable processes can sometimes be stabilized by identifying the possible causes of the instability and taking steps to prevent them from recurring. Processes lacking in capability will require modification and then stabilization, since modification alone is enough to make a process unstable (Florac and Carleton 1999, chap. 7).

Predicting the effort consumed by a process that is out-of-control is like trying to estimate the total duration of a hitchhiking trip. Since hitchhikers do not control the time of pick-up, where drivers are going or how fast they drive, their estimates

will be fuzzy at best. If time is critical, it would be wise to use one's own car or any other more reliable means of transportation. In the same way, if software needs to be developed in a predictable way, then the software process will need to be monitored and controlled to support the desired predictability level.

The use of effort measurement as a basis for process monitoring can bring with it certain advantages. First, effort data are likely to be collected for estimation purposes, so using an effort-based monitoring and control system might reduce or eliminate any additional data collection burden. Second, since effort will ultimately constitute a key measure of project success or failure, using it as an indicator of process control makes a great deal of sense. Third, using effort data to confirm that a software process is under control implies that the same data can theoretically be reused to feed an estimation model.

MacDonell and Shepperd (2003) have studied the effort expended in sixteen software development projects undertaken by a single organization in order to determine the viability of prior-phase effort analysis for re-estimation in the course of a project. They found "little support for the idea of standard proportions of effort distributed between phases." However, their approach involved comparing the effort expended during successive phases of a project. The notion of distinct engineering disciplines applied concurrently towards a succession of phases, as implemented for instance in the Rational Unified Process (Kruchten 2000), was not used in that study. Including the notion of engineering disciplines adds a potentially

useful level of refinement to the problem at hand.

An academic platform has been used for several years for the purpose of recording the effort expended in software development projects. The Software Engineering Studio is both a project-oriented course aimed at senior students in software engineering at the École Polytechnique de Montréal, and a research setting which permits the observation of actual software process activities in the context of the development of a complete software product. The Studio has served as the basis for several empirical studies. Those studies generally focused on a single edition of the Studio and featured effort comparison among teams performing identical projects in parallel. Recently, for instance, Germain and Robillard (2004) found that the cognitive activities performed by students were relatively independent from the software process model used. Also, a set of cognitive activities defined as "active" has been shown to have a nearly constant weight for all six projects under study, regardless of the process used.

This paper presents the analyses of recorded effort that identify relevant effort patterns as well as a new process monitoring model. It shows that it is feasible to analyze projects performed under different processes and development cycles and to extract useful information at the process discipline level. First, we provide a general overview of the Studio and of the projects under study. Second, we present the three disciplines that we use as a comparative basis for all projects, and aggregated effort data for each team project. Third, we add the dimension of

project completion to the dimension of process discipline, and shift our focus to the progression of the effort mix for each team project. We categorize team behaviors according to specific patterns exhibited, and identify a few fundamental tendencies. Finally we synthesize our findings and provide a few general conclusions.

4.2 The Software Engineering Studio

The Software Engineering Studio is an optional, project-oriented course offered during the winter term to senior-year students in computer engineering. Its purpose is to allow students to gain practical experience in software development by participating in a small-scale software development project. Teams of students must develop a complete implementation based on software requirements specifications provided by the instructors. Also, they must follow a specific software engineering process, the UPEDU. Participants thus acquire experience in building a complete operational software product through the disciplines of analysis, design, implementation, testing and management. This project course teaches them the realities of teamwork, milestones and time constraints. As a secondary objective, students become more familiar with a specific application domain or set of technologies. Earlier versions of the Studio are presented in detail in Germain, Dulipovici and Robillard (2002a); Germain, Robillard and Dulipovici (2002b); Germain and Robillard (2003, 2004); Robillard (1998). The Unified Process for Education (UPEDU) which has been used for most of the projects performed in the Studio since 2001

is extensively described in Robillard, Kruchten and d'Astous (2001, 2003) and is readily available online (École Polytechnique de Montréal 2002).

The case study described in this paper covers the three following Studio editions: 2001, 2002 and 2003. Although some general principles apply to the design of all three editions, substantial differences do exist. Projects, technologies, processes, specific pedagogical objectives, coaching style, guidelines to students, instructor attitude, team sizes, and effort recording schemes and tools all varied to some extent from one edition to another. The 2001 edition, for example, featured the development, using Java-related technologies, of a Web-based effort recording tool similar to the one that was used by the participants to carry out their own effort logging activity. The 2002 edition, which is described in detail in Germain and Robillard (2003), featured the development of a Web-based meeting scheduling system. The implementation had to be performed using Java-related technologies and Oracle databases. In 2003, the project would involve performing major maintenance operations on one of the systems developed during the previous year. The technological base remained essentially the same, except for the fact that students were required to make the system work with the latest releases of the Java Runtime Environment.

4.3 Team-projects under study

A total of 15 teams participated in all three editions under study. Only eight team-projects were selected for this comparative analysis because of the quality

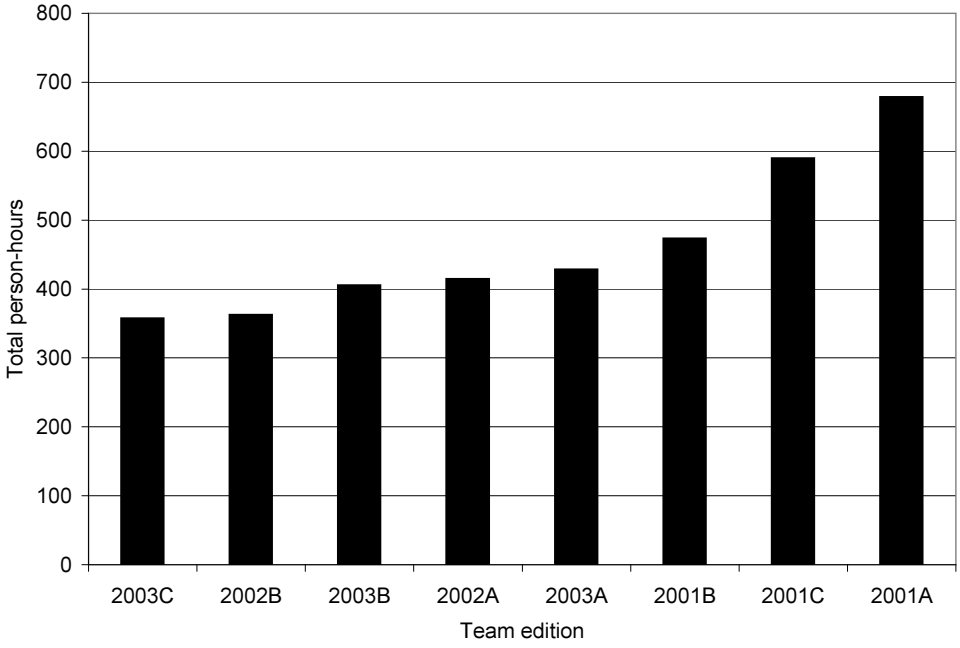

Figure 4.1: Distribution of effort among team-projects

of the deliverable products. The seven team-projects excluded had suffered major drawbacks, such as failure to meet the requirements, unreliable effort slip values, insufficient overall effort, lack of relevant team activities and major misunderstanding of the software engineering process disciplines. All selected team-projects involved a minimum of 350 person-hours. The projects selected for the study generated software system products that met the specified requirements. Figure 4.1 shows the distribution of total effort for each team-project. Project duration varied from 9 to 14 weeks depending on the studio edition. For instance, the first column shows that project C in year 2003 required a total effort of 350 hours.

4.4 Overview of disciplines

The first step is to define the three disciplines to be used as a comparative basis for all the projects. These disciplines must be general enough to fit each of the projects, and specific enough to provide a meaningful description of the various activities composing the discipline. The following three disciplines meet these criteria.

- The "Coding" discipline includes all activities related mainly to programming.
- The "Validation and Verification (V&V)" discipline includes all activities related to debugging, testing and integration of developed components. It does not, however, include reviews of requirements, analysis and design artifacts.
- The "Engineering" discipline includes all activities related to requirements, analysis and design, as well as technical planning at the integration and implementation levels. The various artifact review activities are also included.

Management activities, which include project planning, scheduling, tasks assignments, progress reporting and the like, are not used for this analysis.

Figure 4.2 shows the normalized effort distribution for the eight team-projects selected for the three years studied, sorted by increasing total effort. To enable comparison, the total effort for each team-project is normalized to 100%. Each column represents the relative effort for each discipline for a given team-project. For example, the column at the far left represents the team-project with the smallest

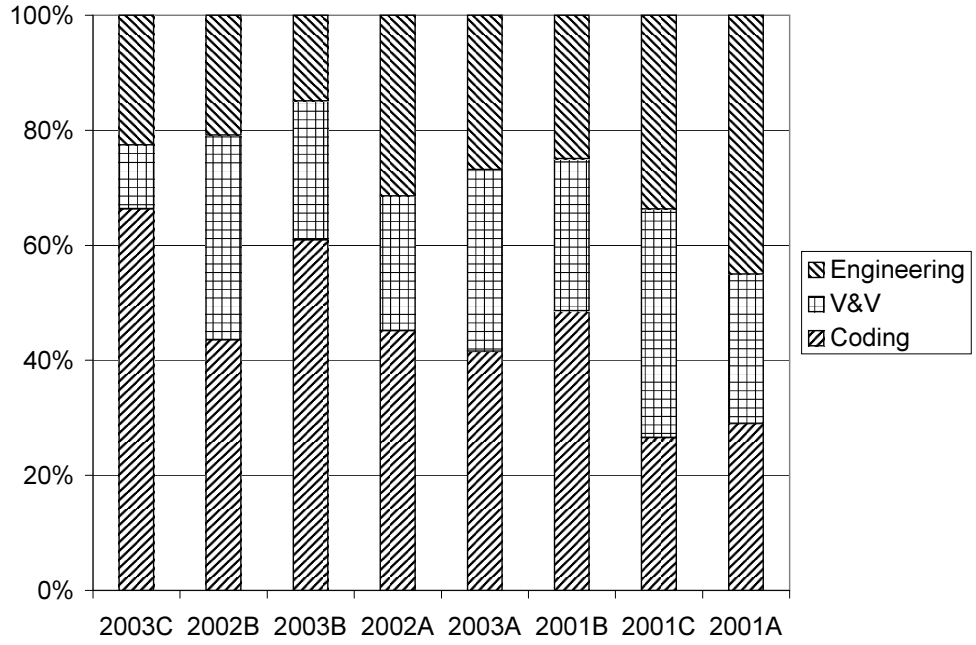

Figure 4.2: Relative effort distribution by category, for all team-projects, sorted by increasing total effort

total effort, and the one that expended the most relative effort in coding activities. Figure 4.2 shows that the relative effort for each discipline varies widely from one team-project to another, and that effort distribution is not necessarily related to a specific project. It also shows that projects carried out in the same year vary widely in relative effort ratios.

4.5 Study of effort distribution invariance relative to project factors

Figure 4.3 shows a 2D plot of team-project positioning relative to the three poles formed by each of the three defined disciplines. Since the relative effort in each discipline for a given team-project adds up to 100%, it is possible to represent those

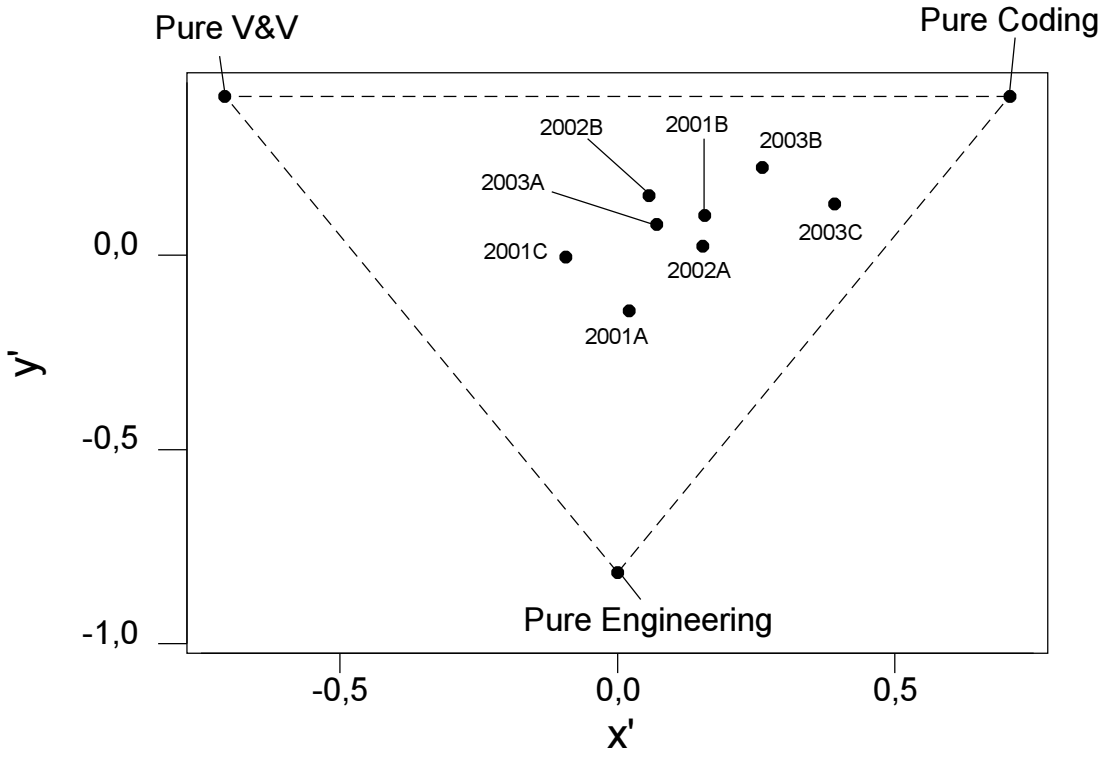

Figure 4.3: 2D plot of total team-project effort ratios relative to the three poles

three variables for each team-project in a plane. The resulting graph is bounded on the three axes by the maximum value for each discipline, which is 100%. The marker at the far right is reached for a team expending all its effort in coding, while the marker at the far left is reached for a team-project expending 100% of its effort on V&V; consequently, the marker at the bottom of the diagram represents full engineering activities. A description of the axes shown, as well as calculation details, can be found in the appendix.

These data points represent final project states and provide no information regarding the evolution of the relative weight of each discipline involved in the

software engineering process. This can be misleading, as the relative weights of disciplines are likely to vary over time (Germain, Robillard and Dulipovici 2002b). We have therefore taken snapshots of the cumulative relative effort at 20%, 40%, 60%, 80% and 100% of project completion.

The next eight figures show 2D plots of the evolution of the position for each team-project in terms of the relative discipline ratios for each 20% of the project's duration. The sequence of point positions represents the evolution of the relative effort by discipline throughout the project. It is important to note that, by nature, these diagrams are cumulative, and, therefore, that the effort slip sample is much lower at, say, 20% than at 100%. Project advancement steps for a given team-project have been defined as the ratio of cumulative effort expended at a given position to the total effort expended in the whole project.

4.5.1 Regular and coherent project progressions

Figures 4.4, 4.5, 4.6 and 4.7 show progression of the discipline ratios for a first group of four similar team-project patterns. The four progression patterns shown are clear, smooth and coherent, although pattern details differ from one team-project to another.

These diagrams show that at the 20% stage in project advancement, the effort was mostly invested in the engineering discipline. At the 40% stage, some V&V and coding effort was added, pushing the data point upward. At the 60% stage, major

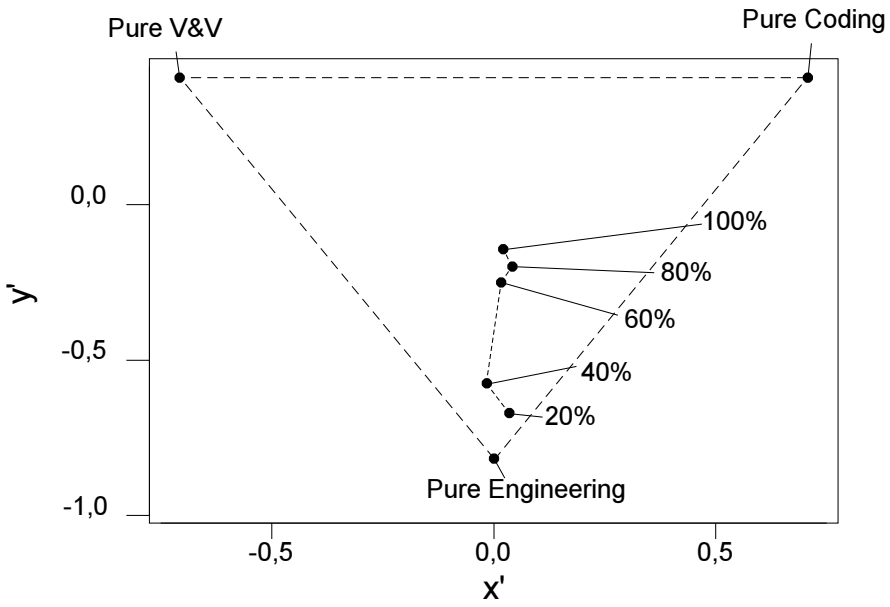

Figure 4.4: 2D plot of effort-mix progression for team-project 2001A

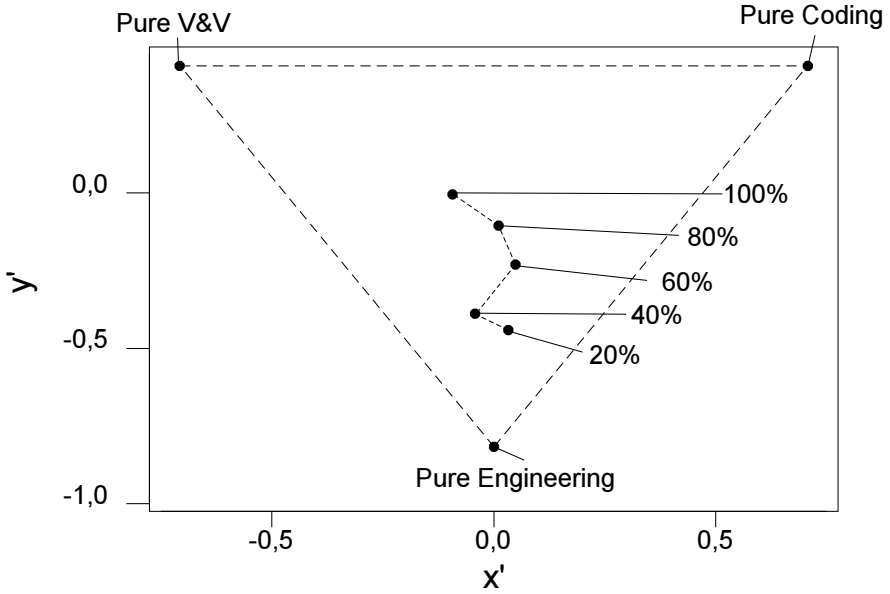

Figure 4.5: 2D plot of effort-mix progression for team-project 2001C

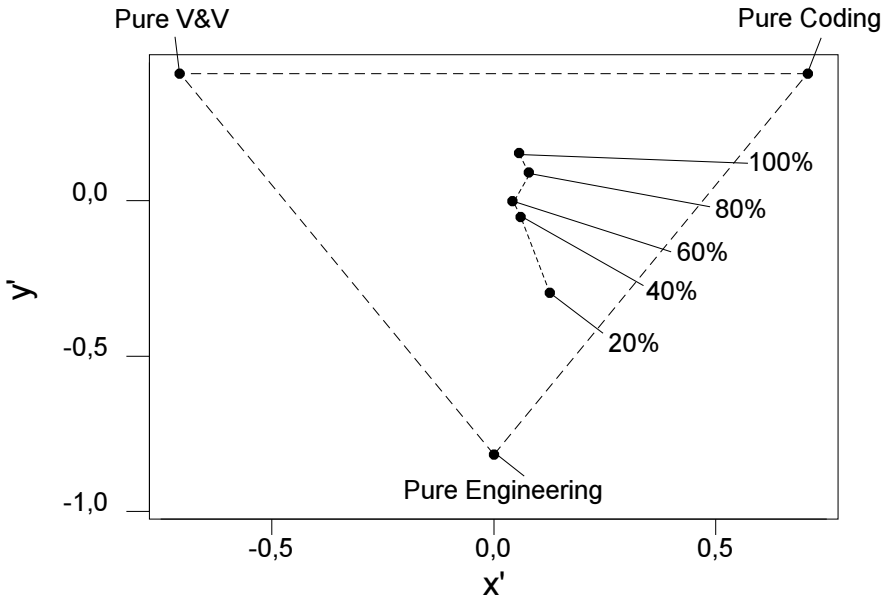

Figure 4.6: 2D plot of effort-mix progression for team-project 2002B

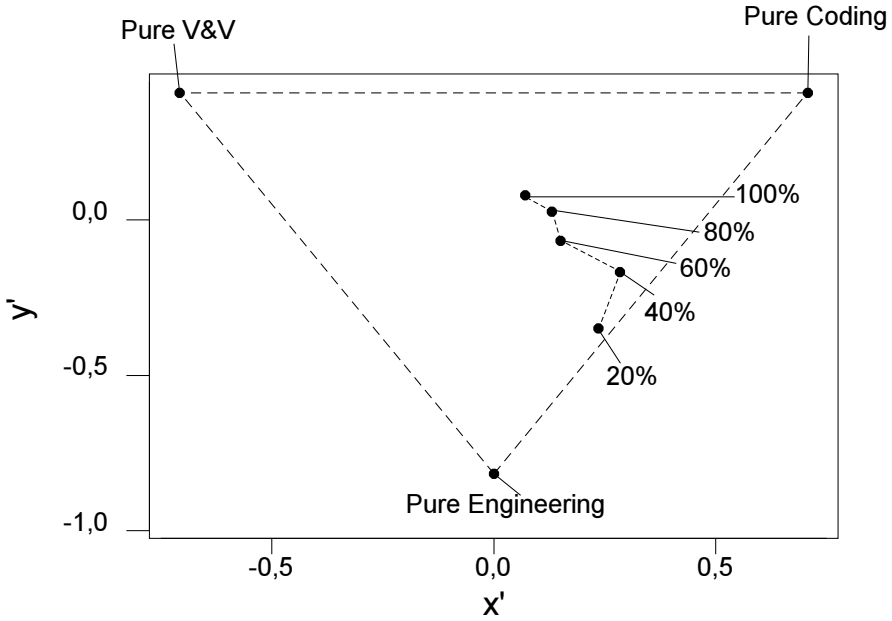

Figure 4.7: 2D plot of effort-mix progression for team-project 2003A

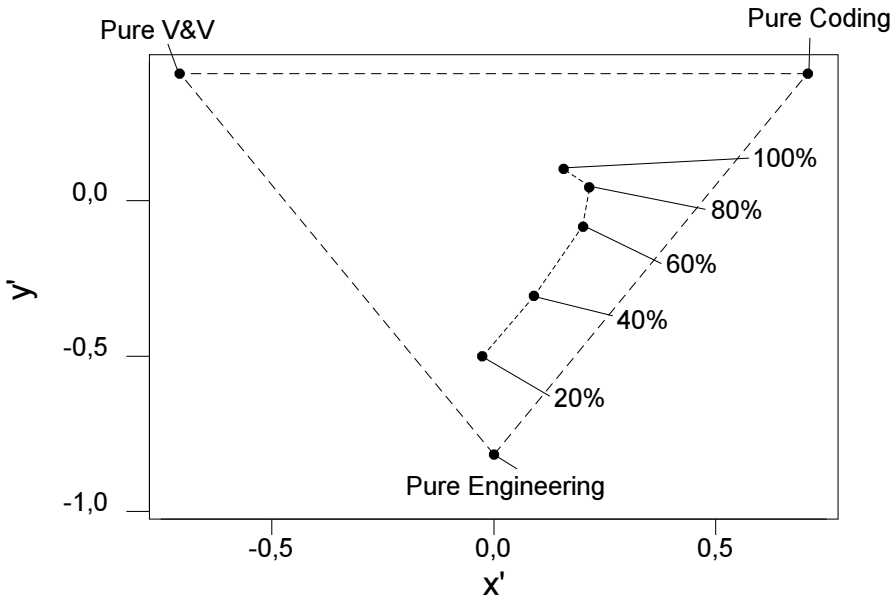

Figure 4.8: 2D plot of effort-mix progression for team-project 2001B

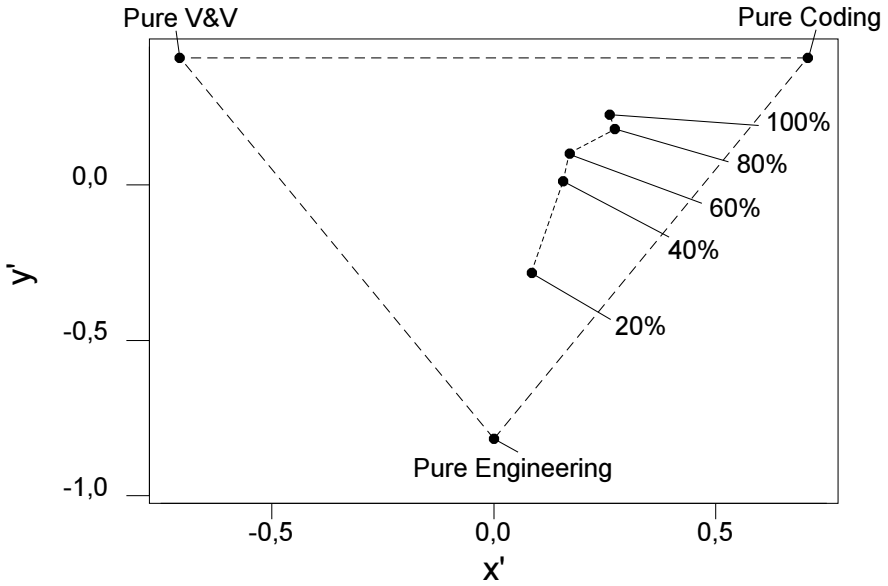

Figure 4.9: 2D plot of effort-mix progression for team-project 2003B

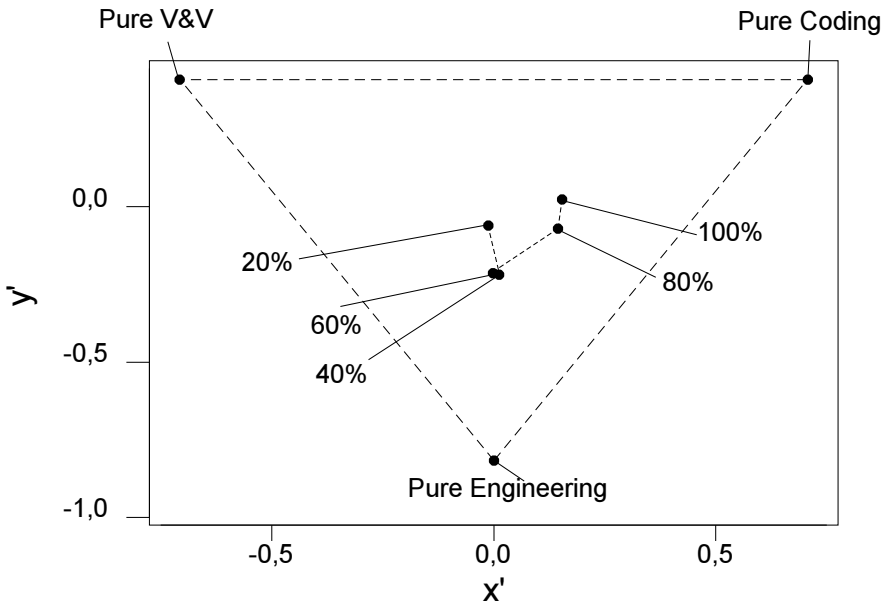

Figure 4.10: 2D plot of effort-mix progression for team-project 2002A

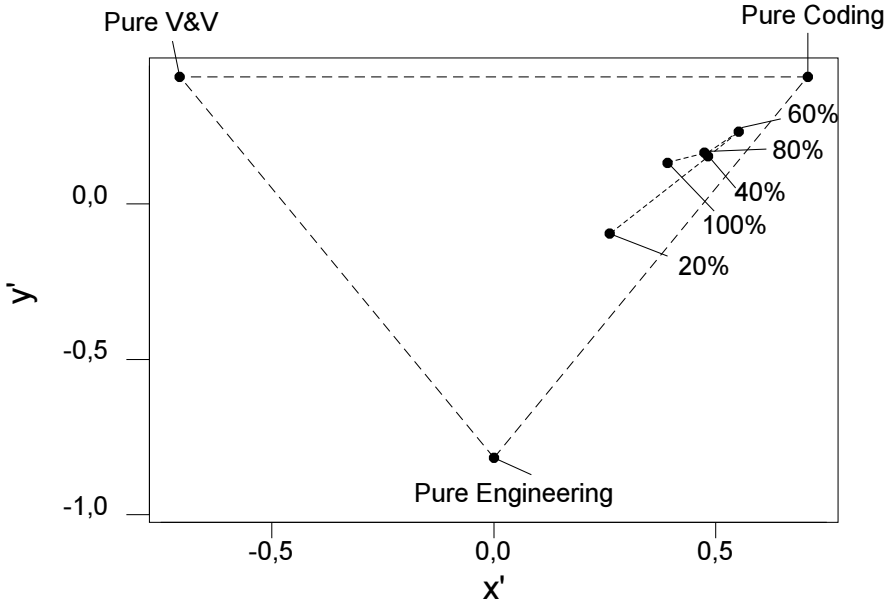

Figure 4.11: 2D plot of effort-mix progression for team-project 2003C

effort was added about equally in the coding and V&V disciplines. Finally, at the 80% and 100% stages, projects end almost in the middle of the diagram, which means that total effort has been invested almost equally among the disciplines. Since the 20% stage is mainly at the engineering marker, we can conclude that the engineering effort was very important for these team-projects.

For these first four team projects, all progressions head "north" (assuming that "north" is at the top of each diagram), meaning that the emphasis is put on engineering activities at the beginning of a project, that their relative importance diminishes as the project progresses, and that, from there on, the coding and V&V activities increase about equally. Team-project 2003A, shown on Figure 4.7, is a slight exception, as progression heads "north-west" rather than "north", which means that most of the engineering effort is transferred to V&V activities as the project progresses. The 20% stage is also very interesting as it includes a very small V&V component as well as a 50-50% mixture of coding and engineering activities. It is interesting to note that at least one team-project from each edition is represented in this subgroup.

4.5.2 Coherent progressions with tilt towards coding

Figures 4.8 and 4.9 show a special case of coherent progression, where little or no increase in V&V activities occurs. This transitioning from mostly engineering activities to mostly coding activities is remarkable because it tends to represent a

break from the accepted practices in iterative and incremental software development.

Determining whether a progression is regular or not can be made only on a case-by-case basis, since there is no obvious demarcation between what constitutes a sufficient increase in V&V activities and what does not. We selected team-projects 2001B and 2003B because they exhibit at least one clear, measurable characteristic of inclination towards coding activities. In the case of team-project 2001B (see Figure 4.8), there is virtually no increase in V&V activities until after the 60% completion point. After that point, the increase is sharp but limited.

In the case of team-project 2003B (see Figure 4.9), V&V activities increase little over the course of the project relative to other activities and actually decrease between 60% and 100% of completion.

4.5.3 Incoherent progressions

Figures 4.10 and 4.11 show two incoherent progressions of the effort mix. In Figure 4.10, the progression shown for team-project 2002A has a "U" shape which starts almost in the middle of the diagram, comes down towards the "Engineering" pole, and then, after a pause, moves towards the "Coding" pole. The incoherence is mostly due to the data point at 20% of completion which includes too few engineering activities. However, were that point to be located closer to the "Engineering" pole, then the overall pattern would be tilted towards the "Coding" pole and thus

still be considered irregular.

In Figure 4.11, we can see that the progression of the effort mix for team-project 2003E evolved in a chaotic fashion along the axis between the "Engineering" and "Coding" poles. Not only is there no real progression, but the proportion of V&V activities through the entire project remains remarkably low.

4.6 Non-cumulative perspective on regular coherent progressions

Figure 4.12 shows the non-cumulative effort patterns for the four team-projects which have exhibited regular coherent effort progressions. The curves are shown on different "floors". The x-axis represents ranges of project advancement as defined previously. The sum of the three curves amounts to 100% for every selected range. Terms "Elaboration", "Construction", "Build-up" and "Finalization" shown at the top of the diagram are discussed in section 4.7.2. The diagram shows that coding effort starts at a low level, increases smoothly to a peak at between 40% and 80% of completion and then comes back to its earlier levels. V&V effort starts at a low level and increases steadily through the entire project, with a plateau at between 40% and 80% of completion. Engineering effort starts at a very high level (69%) and decreases abruptly to a very low level (below 20%), which is maintained from 40% of completion to the end of the project.

Figure 4.13 shows the same information for team-projects with irregular but coherent progressions. As mentioned earlier, irregularity occurs in the case of a very

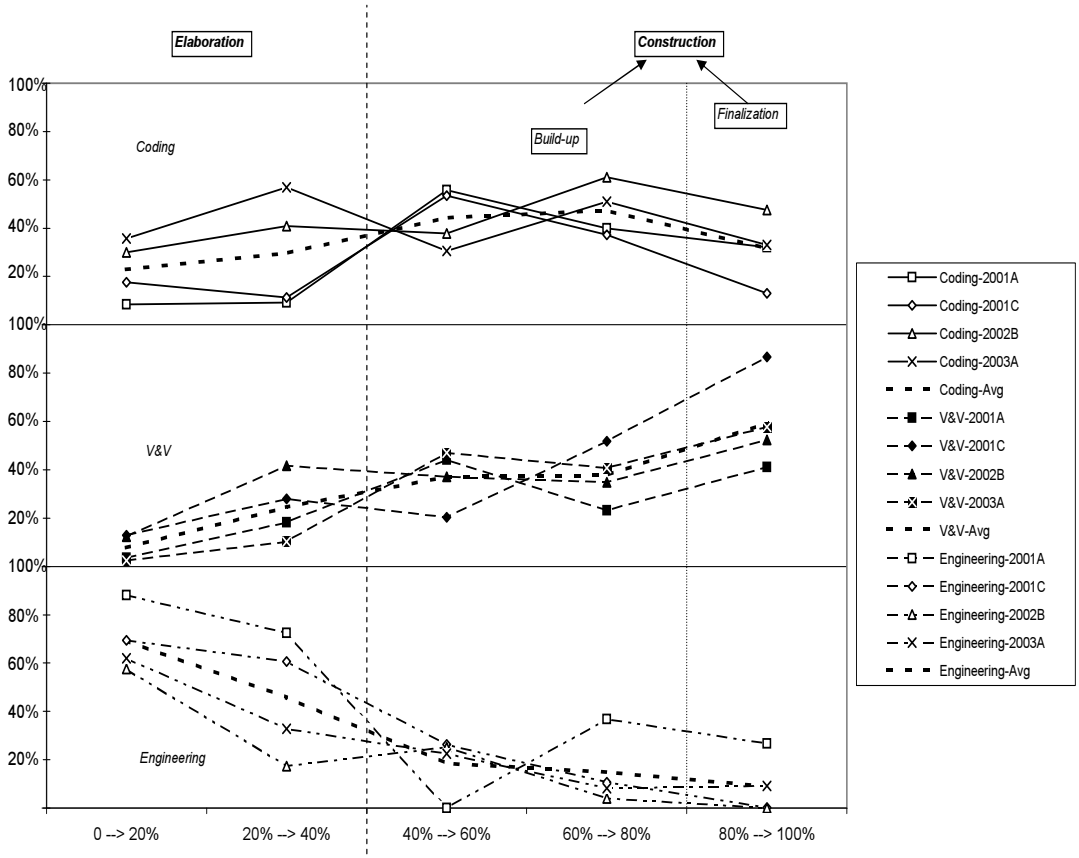

Figure 4.12: Non-cumulative effort for team-projects with regular and coherent progressions as a function of project advancement

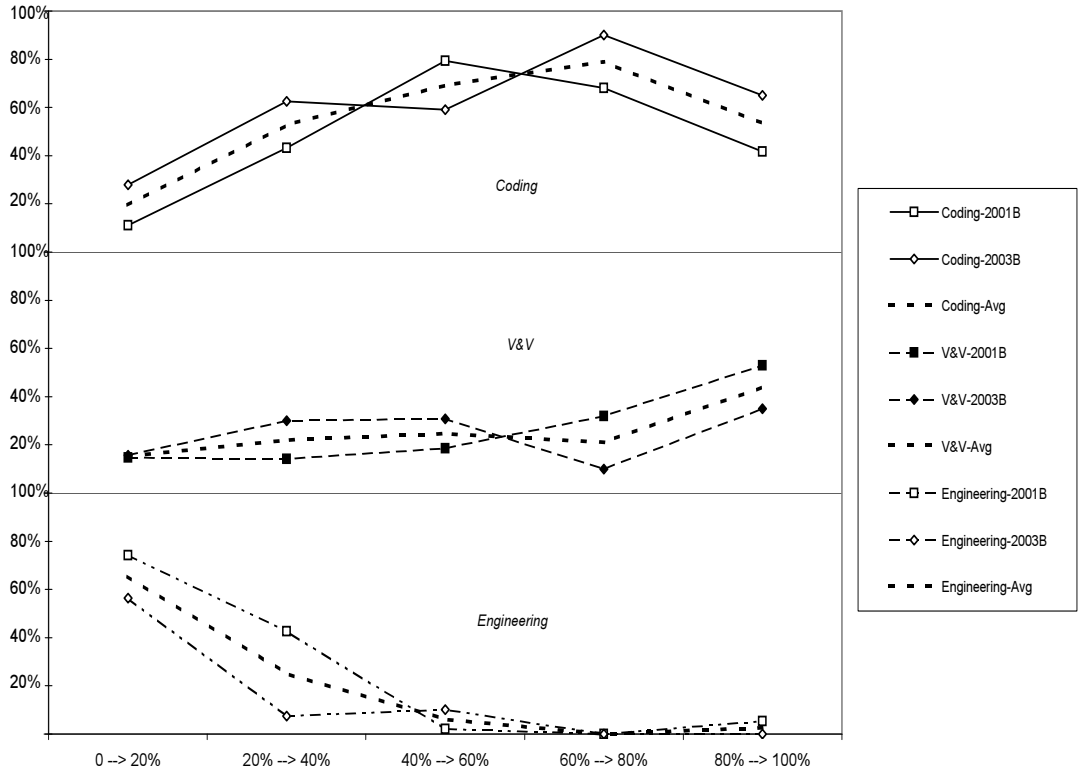

Figure 4.13: Non-cumulative effort for team-projects with irregular but coherent progressions as a function of project advancement

low increase in V&V activities. However, the diagram also shows that, while the patterns of engineering activities are quite similar to those of the team-projects with regular progressions, coding effort curves have much higher peak values, whether on average (79% vs. 47%) or for extrema: the peak for the lower curve in Figure 4.13 is indeed at 79%, while the lowest of the four peaks in Figure 4.12 is at 61%. This particular behavior seems totally independent of project absolute effort values. We can also see that there is indeed some increase in V&V activities in Figure 4.13, but that, as opposed to the increase in Figure 4.12, it does not take place until after 80% of the projects have been completed. These observations tend to show that irregular progressions do not simply constitute a special case of coherent progressions, but rather should be considered as an independent category characterized by its emphasis on coding activities.

Figure 4.14 shows the same information for the team-projects with incoherent progressions. Average values have been omitted as patterns do not converge, except a little for V&V activities. Both team-projects exhibit major deviations from patterns previously shown. For team-project 2002A, the peak in engineering activities is located in the second fifth of project execution instead of the first. Also, V&V activities actually decrease over the course of the project. Finally, the increase in coding activities from 40% to 80% of completion is unusually high. For team-project 2003C, coding and engineering effort patterns show the inverse of what is shown by other team-projects.

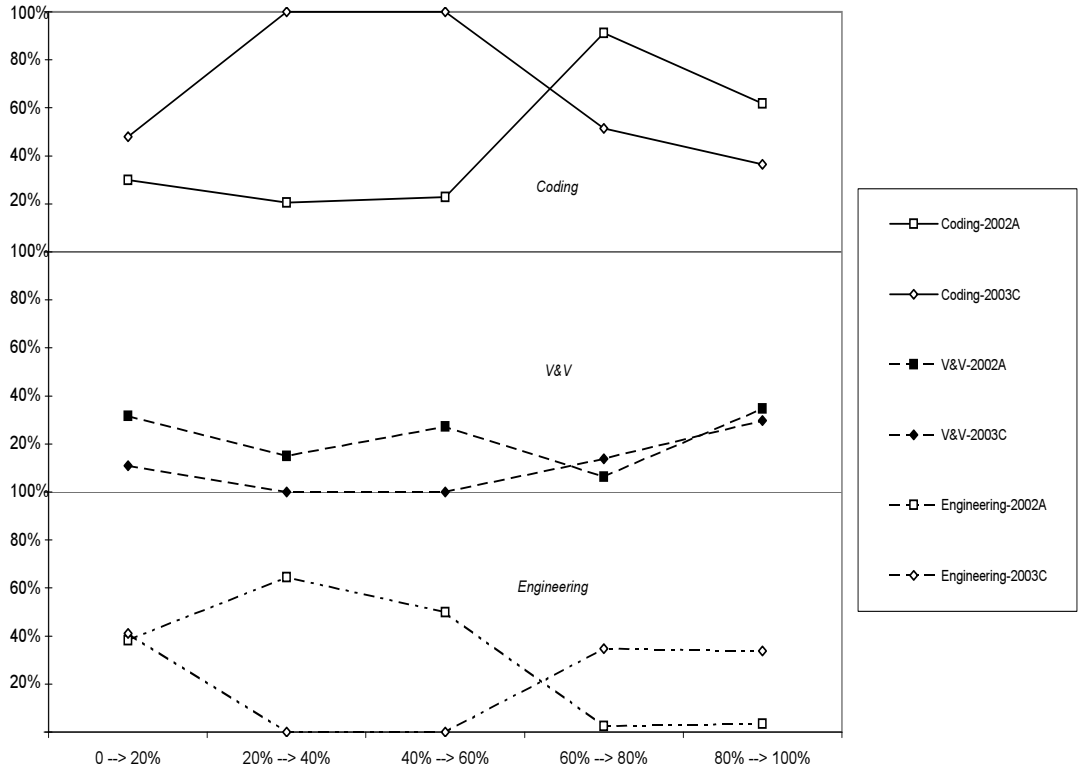

Figure 4.14: Non-cumulative effort for team-projects with incoherent progressions as a function of project advancement

4.7 Analysis

4.7.1 Coherence and regularity

Six of the eight team-projects studied show coherent effort patterns along the three disciplines which constitute the basis of our classification. The focus is placed on engineering activities first, and is gradually transferred to coding and V&V activities. For four of them, the increase in V&V activities is at least as great as it is for coding activities. These observations tend to show the presence of fundamental behavior characteristics. The fact that two team-projects exhibit incoherent progressions of effort patterns should not be a surprise, since the participants in them were mostly inexperienced, and, while the software process gave clear indications of the kinds of activities to perform and artifacts to generate, it did not, as implemented, tell the developers when to perform those activities. Timing issues are tightly coupled to the development cycle implemented, and were, except for delivery dates, mostly under the control of the teams.

4.7.2 Development cycle and key milestones for regular and coherent progressions

The progression of the curves shown in Figure 4.12 offers a picture of a fractured software development cycle. These "fractures" match the "Elaboration" and "Construction" phases of the UPEDU. (Note that the contents of the "Inception"

and "Transition" phases are not included in any of the projects under study, so we do not consider these here). The 40% of project completion point seems to constitute the first major milestone. Before that point, the focus is on engineering activities, which is typical of an "Elaboration" phase, evolving rapidly towards this first milestone. After 40% of project completion, the process seems to become much more stable, with an average of only 14% engineering work in the second half of the software development cycle. The whole time period from the first milestone to the end of the project is typical of a "Construction" phase.

The 80% completion stage seems to represent a second milestone, with a great deal of emphasis being transferred mostly from coding activities to V&V activities. This milestone is less important than the first one, in that a shift from coding to V&V activities has to be expected at the end of a typical construction phase, with the timing of the shift being theoretically dependent on iteration granularity. We call the part of the "Construction" phase that occurs before the 80% milestone the "building-up" stage. This period reflects a steady state which is, in theory, typical of iterative and incremental processes during the time period when components are built and basic system integration is performed. "Finalization," by contrast, is the part of the phase that occurs beyond the 80% milestone. At that point, software system construction focuses on integration and validation issues.

4.7.3 Impacts on effort estimation and process control

These observations open the way to substantial improvement in the way we perform process monitoring in software engineering. Thus, it should be possible to determine the appropriate effort mix at any given time. Major discrepancies from accepted patterns may signal a process which is out of control. This subject is discussed extensively in Münch and Heidrich (2004); Selby, Porter, Schmidt and Berney (1991). In particular, identifying such situations and applying countermeasures are tasks which are likely to be much more effective using an approach by discipline than using a single indicator of cost overrun, as problematic situations may be identified earlier in the course of the project.

One key aspect of such an approach is determining the actual state of project advancement at a given time. A few distinct approaches may be used, depending on the variables that have generated the most confidence. For instance, under some circumstances, project advancement may be estimated by artifacts, such as architecture documents, which can then be used to compare actual and theoretical curves for that completion level. Conversely, project advancement at a given time could possibly be calculated by analyzing effort patterns as drawn up to that point, leading to re-estimation of total project effort.

Of course, the limitations applicable to estimation techniques (Boehm, Clark, Horowitz, Westland, Madachy and Selby 1995) apply here too. First, estimation will always become more accurate as the project advances, whatever techniques are

used. Second, databases of historical data will continue to be required in order to adapt to local project and team conditions.

The main contribution of this paper is to show that process monitoring and control based on effort patterns and on a categorization by discipline is likely to be a practical approach, and that the proposed discipline scheme has very good potential to achieve that goal. This is not a trivial issue, as other categorization schemes may not necessarily lead to converging patterns. Germain and Robillard (2004) tested two different categorizations while analyzing patterns of cognitive activities performed in the 2002 edition, and found that one of them was much more sensitive to the process used. Other potentially interfering factors include individual team development style, which remains extremely difficult, if not impossible, to control without implementing a software process that is very restrictive.

4.7.4 Study limitations

Limitations applicable to this kind of study, and already mentioned in Germain and Robillard (2004), apply here too. Thus, as discussed earlier, student developers' behavior could be substantially different from professionals' behavior. This shows that our observations should be generalized with great care. Also, the validity of the study rests heavily on the level of confidence that exists in the effort slips entered by the participants. However, the larger number of projects analyzed in the present study, as well as the remarkable convergence of effort-mix patterns between

team-projects, contributes to attenuating, but not eliminating, that factor.

4.7.5 Effort distribution: does it make sense?

This study shows that the coding discipline covers, on average, 45% of total effort, and that direct component implementation activities account for approximately two-thirds of that percentage. Do these figures make sense? A study performed by the Bangalore SPIN (2001) on projects averaging 30,000 lines of code has shown that coding and unit-testing activities amounted to approximately 42% of the effort devoted to non-management tasks. Meanwhile, a technical report by the Davis and Mullaney (2003) shows that, for a sample of projects in which the Team Software Process (TSP) was used (Humphrey 2000), effort devoted to coding, compiling, performing test set-ups and unit-testing amounts to 40% of the total, or 42% if we exclude planning and post-mortem work. These comparisons allow us to have some confidence, albeit in a limited way, in the relationship between the behavior of software engineering students in small projects and that of more experienced developers in larger projects.

4.8 Conclusion

This paper has presented an analysis approach for extracting relevant information on effort at the process discipline level and for identifying patterns which are followed in a substantial number of the projects under study. We have been able

to provide quantitative measures of effort progression for projects that are, in our opinion, quite similar to many other small-scale software development initiatives. Such quantification is necessary if we wish to be able to use detailed process information for planning purposes, instead of aggregated information alone. We have also defined a technique which permits the online detection of irregular project progressions using measurements of the effort mix at various times. In order to establish whether or not our proposed progression is ideal for that purpose, it will have to be validated using additional empirical data similar as that presented here. The main contribution of the paper is to show that an opportunity currently exists for the useful utilization of effort progression data for project planning, monitoring and control purposes.

Acknowledgments

We are grateful to Sébastien Cherry, Alexandre Moïse, Christian Robidoux, Martin Robillard and Houcine Skalli who were involved at various steps in the three Software Engineering Studio editions, and especially to Mihaela Dulipovici who acted as teaching assistant for editions 2002 and 2003 and was deeply involved in artifact and effort-slip evaluation in all three editions. Also, this study would not have been possible without the participation of all the students enrolled in the three editions.

This work was partly supported by the Natural Sciences and Engineering Re-

search Council of Canada (NSERC) under grant A0141.

CHAPITRE 5

DISCUSSION GÉNÉRALE

De nombreux éléments de discussion ont été présentés à l'intérieur des deux chapitres précédents. La présente section vient enrichir ces éléments par quelques considérations d'ordre général.

5.1 Calibration des charges de travail

Le contenu des spécifications émises a été calibré de manière à refléter une charge de travail compatible avec un cours de trois crédits, soit l'équivalent de 135 heures par étudiant. En l'absence de données à long terme qui auraient pu alimenter un modèle d'estimation en bonne et due forme, cette calibration a été effectuée sur la base du jugement des enseignants. Selon la recherche effectuée, on pose l'hypothèse que la charge de travail attribuée aux équipes était raisonnable compte tenu des ressources et du temps alloués pour la réalisation des projets. À titre d'illustration, deux modèles souvent utilisés pour fins d'estimation, soit le COCOMO II et le modèle SLIM de Putnam, tiennent pour acquis que la relation entre la durée optimale de réalisation D, typiquement en mois, et l'effort requis E, typiquement en personnes-mois, n'est pas linéaire (Fenton et Pfleeger 1997),

suivant généralement une courbe de la forme

$$D = aE^b$$

où a et b sont des constantes et $0 < b < 1$. Dans University of Southern California (1998), il est mentionné que :

> "(...) Accelerated schedules tend to produce more effort in the later phases of development because more issues are left to be determined due to lack of time to resolve them earlier. (...) A stretch-out of a schedule produces more effort in the earlier phases of development where there is more time for thorough planning, specification and validation."

En pratique, la convergence remarquable des patrons d'effort présentés au chapitre 4 peut porter à croire que la présence éventuelle de tels phénomènes a eu peu d'impact sur la qualité des observations. Incidemment, la mesure de l'avancement des projets a été réalisée en utilisant la proportion de l'effort total réalisé à un instant donné, plutôt que les dates indiquées sur les jetons d'effort. Ce choix avait justement pour but de minimiser l'impact de ces phénomènes.

5.2 La classification en disciplines

La classification des activités cognitives ou de processus en catégories ou disciplines a été réalisée en fonction des contraintes inhérentes aux processus et aux

approches de mesure de l'effort utilisés. Bien qu'ayant permis la réalisation d'observations intéressantes, les classifications réalisées sont exploratoires et devront être raffinées. La réalisation de classifications pertinentes est essentielle pour l'obtention de patrons significatifs et, partant, pour le développement de mécanismes viables en matière de contrôle de processus et d'estimation en cours de projet. Les principales caractéristiques de qualité d'une classification en disciplines sont les suivantes :

- La classification doit pouvoir s'arrimer naturellement avec la ou les classifications de bas niveau des activités cognitives ou de processus utilisées dans le cadre de la prise de mesures. Idéalement, chaque activité doit pouvoir être rattachée clairement à l'une ou l'autre des disciplines, sans qu'il soit nécessaire d'étudier le contexte du jeton avant de pouvoir prendre une décision à cet effet.
- La classification doit permettre une discrimination claire entre les courbes d'effort issues de chaque discipline.
- Elle doit pouvoir absorber des variations raisonnables sur le plan de facteurs tels que le processus utilisé, la nature ou la taille des projets, ou encore la taille des équipes, sans que ces variations ne se reflètent de manière significative dans les courbes d'effort. Le caractère raisonnable des variations acceptées dépend du contexte de la mesure et de l'utilisation prévue des résultats.
- La classification doit comporter un nombre optimal de disciplines et ce, afin d'équilibrer les deux contraintes suivantes :

- Elle doit permettre l'élimination des effets découlant de la substitution mutuelle possible des activités enregistrées par les participants. De telles substitutions peuvent survenir notamment pour des raisons d'ambiguïté de la classification de bas niveau des activités, ou encore pour des raisons de choix de méthodes de travail non couvertes par la classification de bas niveau des activités.
- Elle doit comporter un nombre suffisant de catégories afin d'offrir une perspective fine lors de l'analyse des patrons d'effort. Un nombre élevé de patrons significatifs est suceptible d'engendrer une meilleure précision d'analyse.

Le principal avantage de la classification par état employée au chapitre 3, relativement à la génération de patrons d'effort, est la stabilité montrée par rapport au modèle de processus utilisé. En pratique, toutefois, l'utilité de cette caractéristique peut varier considérablement selon le contexte. Une organisation désireuse de pratiquer le contrôle de processus et disposant d'un modèle générique de processus qu'elle réussit à appliquer avec peu de modifications à la plupart de ses projets risque peu d'en bénéficier. De façon générale, une classification sera plus utile si elle réussit à soutenir des conditions de projets diversifiées plutôt que des processus très différents. De toute manière, des conditions de projets variables devraient normalement entraîner des variations correspondantes dans les modèles de processus utilisés. La classification par état comporte par ailleurs l'inconvénient d'être diffici-

lement utilisable à l'extérieur d'un contexte où les données d'effort sont recueillies selon les activités cognitives réalisées. En effet, une activité de processus est susceptible d'incorporer à la fois des activités cognitives liées à la réflexion, à l'action et à l'interaction. Ces éléments ne doivent toutefois pas occulter l'utilité remarquable de cette classification à des fins d'analyse empirique, ainsi que les conclusions du chapitre 3 le laissent entrevoir.

La classification par étape possède une structure similaire à la classification en trois disciplines présentée au chapitre 4. Les deux classifications séparent également les activités réalisées selon leur position par rapport à un noyau central, à savoir l'implantation des composants logiciels. Les activités en aval de ce noyau sont placées dans une catégorie ; celles en aval sont placées dans une autre, et les activités d'implantation sont placées dans une troisième. Les différences dans la répartition des activités au sein de chacune des catégories tiennent avant tout au rôle des classifications dans leur étude respective. La classification originale par étape avait notamment été conçue avec l'idée d'isoler spécifiquement les activités de préparation et de contrôle, supposément davantage exécutées en mode UPEDU qu'en mode XP (ce qui a été confirmé dans un seul des deux cas). Bien que cet objectif s'appliquait dans une moindre mesure à la classification du chapitre 4, la préoccupation de base dans ce dernier cas était plutôt de cerner trois pôles d'activités représentant des modes de fonctionnement enclenchés successivement dans l'exécution d'un projet typique. Nonobstant ces différences, la structure commune

TAB. 5.1 – Un exemple de classification en disciplines, basée sur les disciplines originales du UPEDU

	Préparation	*Implantation*	*Contrôle*
Ingénierie	Requis, analyse et conception (sauf les revues)	Implantation (planif. technique de l'intégration et de l'implantation)	Revues (Requis, analyse et conception)
Programmation		Programmation	Revues de programmation
Validation et vérification (V&V)		Implantation (débogage, tests unitaires et intégration) Tests	

aux deux classifications rend possible l'étude d'une classification plus fine basée sur la juxtaposition des deux classifications présentées. Le tableau 5.1 montre un exemple de classification en sept catégories suivant ce modèle, basée sur les disciplines du UPEDU. Bien entendu, une telle classification reste à valider dans le cadre d'une démarche empirique, tel que discuté ci-après.

CONCLUSION

Ce travail de recherche a permis d'approfondir notre compréhension des relations entre les caractéristiques des processus et des projets de développement de logiciels et le comportement des individus qui participent à ces projets. Au-delà de la rhétorique ambiante sur la valeur comparée des multiples approches de développement, il ressort que le choix d'une approche particulière a un impact limité sur les activités cognitives réalisées par les équipes étudiées. Cet impact, mesuré en termes relatifs, se manifeste surtout au niveau des activités d'inspection et de révision des travaux réalisés.

L'analyse des activités réalisées par chaque équipe étudiée montre une convergence modérée des proportions d'activités réalisées sur l'ensemble du cycle de développement. Toutefois, l'observation la plus notable demeure que la majorité des projets soumis affichent une évolution très similaire du type d'activités réalisées au fil du temps. Ainsi, l'effort réalisé suit une progression qui débute par une première phase, où les activités liées à l'ingénierie sont prédominantes. Le poids de ces activités diminue progressivement, ce qui amène une seconde phase d'équilibre entre les activités de programmation et celles de validation et de vérification. Cet équilibre est rompu peu avant la fin du projet, en faveur des activités de validation et de vérification, quoique de manière peu marquée. Bien que l'ensemble de ces observations soit en accord avec notre compréhension théorique des cycles de

développement, l'étude réalisée permet à la fois de confirmer la forte convergence des patrons d'effort affichés et de les quantifier en termes relatifs à l'effort total déployé pour chaque projet.

Un des apports principaux de ce travail de recherche est une méthode permettant de pallier à la problématique soulevée par MacDonell et Shepperd (2003), qui n'ont pu relever clairement de « proportions normalisées » d'effort pour chacune des phases du cycle de développement. Notre approche montre que de telles proportions peuvent apparaître, à condition de considérer deux dimensions d'analyse (les disciplines ainsi que l'avancement des projets) plutôt qu'une seule (la notion de phase). Les observations effectuées montrent également que l'utilisation de conditions multiples de projets peut permettre de cerner des patrons d'effort pertinents.

Selon Tichy (2000), « The reality of even the most rigorous approach to empirical work is that experiments normally constitute only a small step forward. ». Cette déclaration s'applique tout à fait au présent travail, d'autant plus qu'il s'agit ici d'une recherche exploratoire et non d'une expérimentation. En ce sens, seule la recherche future fondée sur ce travail permettra de le faire fructifier. Ainsi, nous décelons un certain nombre de voies à suivre pour valider nos observations dans le cadre d'une démarche expérimentale formelle et vérifier leur applicabilité aux fins de la planification, de l'estimation et du contrôle des projets et des processus de développement de logiciels. À cet égard, les recommandations suivantes devraient être prises en considération.

Sur le plan pratique, la validation de l'approche présentée dans le cadre d'une étude portant sur un nombre élevé de projets industriels se pose comme une perspective de recherche immédiate. Elle peut être réalisée en transposant l'approche en milieu industriel, ou encore en mesurant directement la viabilité de la classification en trois pôles dans un contexte d'estimation en cours de projet, comme ce fut le cas dans l'étude de MacDonell et Shepperd en ce qui concerne leur modèle d'analyse par phase du cycle de développement.

Dans une perspective plus théorique, les trois catégorisations présentées ont été analysées à la lumière d'un nombre restreint d'observations. Les caractéristiques observées, notamment l'équivalence de l'effort de la catégorie « Active » pour les deux modèles de processus observés, pourraient faire l'objet d'une tentative de validation dans un contexte plus général.

Une autre avenue de recherche pertinente consiste à tenter de raffiner notre classification en subdivisant les activités en un nombre plus élevé de disciplines. Les trois classifications présentées comportent chacune trois disciplines. Bien que les choix effectués permettent d'obtenir des résultats intéressants sur le plan théorique, l'élaboration de méthodologies d'estimation et de contrôle en cours de projet risque de nécessiter une granularité plus fine des disciplines. D'autres approches de segmentation, comme celle présentée au chapitre 5, pourraient être proposées et vérifiées.

Par ailleurs, l'utilisation de boucles de rétroaction en matière d'estimation et

contrôle de projets ne passe pas uniquement par une catégorisation précise en disciplines. Les algorithmes et heuristiques de contrôle des processus à partir des informations générées en ligne sont susceptibles d'avoir un impact important sur la stabilité du processus ainsi que sur la qualité et la convergence des estimations. L'obtention de catégorisations valables ouvre donc la voie à une démarche d'améliorations mutuelles et continues des méthodologies touchant l'analyse de l'effort et de celles touchant la rétroaction à apporter.

Une limitation notable de notre approche est l'utilisation de mesures normalisées de l'effort. L'effort normalisé constitue sans aucun doute une approche valable pour l'utilisation de modèles à des fins de contrôle de processus, étant donné l'impossibilité de tenir compte à l'avance des facteurs d'échelle liés à la taille du système à développer. Toutefois, l'analyse de l'effort absolu par des outils statistiques appropriés serait susceptible de faire ressortir la présence de tendances non détectées par notre approche principalement observationnelle. Une telle analyse nécessiterait l'utilisation d'un échantillonnage beaucoup plus important que celui présenté ici.

Finalement, les dernières recommandations ont trait à la méthodologie de mesure proprement dite. D'abord, le contexte de saisie de l'effort (outil, modalités d'utilisation, subdivision des activités, approche par activités cognitives ou par activités de processus, etc.) peut avoir une influence significative sur les données d'effort enregistrées par les participants. Une avenue de recherche pertinente consisterait à déterminer le niveau d'objectivité d'une approche de mesure donnée et à

relever les altérations potentielles apportées aux informations enregistrées ainsi que leurs causes. Le processus de validation des jetons d'effort devrait également constituer l'objet d'une recherche approfondie. Bien que des critères objectifs existants permettent de guider le travail de la personne responsable de la validation, il existe un besoin pour l'établissement d'un processus de validation formel, assujetti aux mêmes critères de qualité que les processus de développement de logiciels dont nous avons traité.

BIBLIOGRAPHIE

AGILE ALLIANCE. 2003. «Agile alliance web site». [En ligne]. http ://www.agilealliance.org (Page consultée le 21 mars 2004).

AMBLER, Scott W. 2002. «AM throughout the XP lifecycle». [En ligne]. http ://www.agilemodeling.com/essays/agileModelingXPLifecycle.htm (Page consultée le 21 mars 2004).

ANTÓN, Annie I. 2003. «Successful software projects need requirements planning». *IEEE Software*. 20:3. 44–47.

BANGALORE SPIN. 2001. «Benchmarking of software engineering practices at high maturity organizations». [En ligne]. http ://www.bangaloreit.com/html/itscbng/Softengcon2001/B2SIG.pdf (Page consultée le 21 mars 2004).

BECK, Kent. 1999. «Embracing change with Extreme Programming». *IEEE Computer*. 32:10. 70–77.

BECK, Kent. 2000. *Extreme Programming Explained : Embrace Change*. Boston : Addison Wesley. 190 p.

BECK, Kent, BOEHM, Barry. 2003. «Agility through discipline : A debate». *IEEE Computer*. 36:6. 44–46.

BLAKE, M. Brian, CORNETT, Todd. 2002. «Teaching an object-oriented software development lifecycle in undergraduate software engineering education».

Proceedings of the 15th Conference on Software Engineering Education and Training (CSEET'02). Los Alamitos : IEEE Computer Society. P. 234–240.

BOEHM, Barry, CLARK, Bradford, HOROWITZ, Ellis, WESTLAND, Chris, MADACHY, Ray, SELBY, Richard. 1995. «Cost models for future life cycle processes : Cocomo 2.0». *Annals of Software Engineering*. 1. 57–94.

BOEHM, Barry W. 1988. «A spiral model of software development and enhancement». *IEEE Computer*. 21:5. 61–72.

BROOKS, Jr., Frederick P. 1987. «No silver bullet : Essence and accidents of software engineering». *IEEE Computer*. 20:4. 10–19.

BROOKS, Jr., Frederick P. 1995. *The Mythical Man-Month : Essays on Software Engineering*. Anniversary ed. Boston : Addison Wesley Longman. 322 p.

COCKBURN, Alistair. 2002. *Agile Software Development*. Boston : Addison Wesley. 278 p.

COCKBURN, Alistair, WILLIAMS, Laurie. 2001. «The costs and benefits of pair programming». *Extreme Programming Examined*. Sous la direction de Giancarlo Succi et Michele Marchesi. Boston : Addison Wesley. P. 223–247.

CUGOLA, Gianpaolo, GHEZZI, Carlo. 1998. «Software processes : A retrospective and a path to the future». *Software Process – Improvement and Practice*. New York : Wiley InterScience. P. 101–123.

D'ASTOUS, Patrick. 1999. *Approche de mesure et d'analyse des activités coopératives lors de réunions de révision technique du processus de génie logiciel*. 219 p. Thèse de doctorat en génie électrique, École Polytechnique de Montréal.

DAVIS, Noopur, MULLANEY, Julia. 2003. *The Team Software Process^{SM} (TSP^{SM}) in Practice : A Summary of Recent Results*. Pittsburgh : Carnegie Mellon Software Engineering Institute. 88 p. CMU/SEI-2003-TR-014.

ÉCOLE POLYTECHNIQUE DE MONTRÉAL. 2001. *Annuaire de l'Ecole Polytechnique : Baccalauréat en Ingénierie*. Montréal : École Polytechnique.

ÉCOLE POLYTECHNIQUE DE MONTRÉAL. 2002. «UPEDU». [En ligne]. http ://www.yoopeedoo.org (Page consultée le 21 mars 2004).

EL EMAM, Khaled. 2001. «Software engineering process». *Guide to the Software Engineering Body of Knowledge : Trial Version (Version 1.00)*. Sous la direction de Pierre Bourque et Robert Dupuis. Los Alamitos : IEEE Computer Society. P. 137–154.

FENTON, Norman E., PFLEEGER, Shari Lawrence. 1997. *Software metrics : a rigorous and practical approach*. London : PWS Publishing. 638 p.

FLORAC, William A., CARLETON, Anita D. 1999. *Measuring the software process : statistical process control for software process improvement*. Reading : Addison Wesley. 250 p.

FUGETTA, Alfonso. 2000. «Software process : A roadmap». *The Future of Soft-*

ware Engineering 2000 : 22nd International Conference on Software Engineering. Sous la direction de Anthony Finkelstein. New York : Association for Computing Machinery. P. 25–34.

GERMAIN, Éric, DULIPOVICI, Mihaela, ROBILLARD, Pierre N. 2002a. «Measuring software process activities in student settings». *Proceedings of the 2nd ASERC Workshop on Quantitative and Soft Computing Based Software Engineering*. Edmonton : Alberta Software Engineering Research Consortium. P. 44–49.

GERMAIN, Éric, ROBILLARD, Pierre N. 2003. «What cognitive activities are performed in student projects». *Proceedings of the 16th Conference on Software Engineering Education and Training (CSEET'03)*. Los Alamitos : IEEE Computer Society. P. 224–231.

GERMAIN, Éric, ROBILLARD, Pierre N. 2004. «Engineering-based processes and agile methodologies for software development : A comparative case study». *The Journal of Systems and Software*. Accepté pour publication.

GERMAIN, Éric, ROBILLARD, Pierre N., DULIPOVICI, Mihaela. 2002b. «Process activities in a project based course in software engineering». *32nd ASEE/IEEE Frontiers in Education Conference*. Piscataway : Institute of Electrical and Electronics Engineers. P. S3G–7–S3G–12.

GLASS, Robert L. 2001. «Agile versus traditional : Make love, not war !» *Cutter IT Journal*. 14:12. 12–18.

HALLING, Michael, ZUSER, Wolfgang, KÖHLE, Monika, BIFFL, Stefan. 2002. «Teaching the Unified Process to undergraduate students». *Proceedings of the 15th Conference on Software Engineering Education and Training (CSEET'02)*. Los Alamitos : IEEE Computer Society. P. 148–159.

HENDERSON-SELLERS, Brian. 2000. «The OPEN framework for enhancing productivity». *IEEE Software*. 17:2. 53–58.

HIGHSMITH, James A. 2000. *Adaptive Software Development : A Collaborative Approach to Managing Complex Systems*. New York : Dorset House Pub. 358 p.

HIGHSMITH, James A. 2002. *Agile Software Development Ecosystems*. Boston : Addison Wesley. 404 p.

HÖST, Martin. 2002. «Introducing empirical software engineering methods in education». *Proceedings of the 15th Conference on Software Engineering Education and Training (CSEET'02)*. Los Alamitos : IEEE Computer Society. P. 170–179.

HUMPHREY, Watts S. 1997. *Introduction to the Personal Software Process*. Reading : Addison Wesley Pub. 278 p.

HUMPHREY, Watts S. 2000. *Introduction to the Team Software Process*. Reading : Addison Wesley. 463 p.

JEFFRIES, Ron, ANDERSON, Ann, HENDRICKSON, Chet. 2001. *Extreme Programming Installed*. Boston : Addison Wesley. 265 p.

KRUCHTEN, Philippe. 2000. *The Rational Unified Process : An Introduction*. 2nd ed. Reading : Addison Wesley. 298 p.

KRUCHTEN, Philippe. 2001. «Agility with the RUP». *Cutter IT Journal*. 14:12. 27–33.

LARMAN, Craig, BASILI, Victor R. 2003. «Iterative and incremental development : A brief history». *IEEE Computer*. 36:6. 47–56.

LEHMAN, Meir Manny. 1987. «Process models, process programs, programming support». *Proceedings, 9th International Conference on Software Engineering*. New York : IEEE Computer Society Press. P. 14–16.

LEHMAN, Meir Manny. 1997. «Process modelling – where next». *Proceedings, 19th International Conference on Software Engineering*. New York : The Association for Computing Machinery. P. 549–552.

LI, Ningda R., ZELKOWITZ, Marvin V. 1993. «An information model for use in software management estimation and prediction». *Proceedings of the second International Conference on Information and Knowledge Management*. New York : ACM Press. P. 481–489.

LUTZ, Raymond P. 1992. «Discounted cash flow techniques». *Handbook of Industrial Engineering*. Sous la direction de Gavriel Salvendy. 2nd ed. New York : Wiley. P. 1289–1314.

MACDONELL, Stephen G., SHEPPERD, Martin J. 2003. «Using prior-phase

effort records for re-estimation during software projects». *Proceedings of the Ninth International Software Metrics Symposium (METRICS'03)*. Los Alamitos : IEEE Computer Society. P. 73–85.

MCCONNELL, Steve. 2002. «I know what I know». *IEEE Software*. 19:3. 5–7.

MILLS, Harlan D. 1980. «The management of software engineering part I : Principles of software engineering». *IBM Systems Journal*. 19:4. 414–420.

MORISIO, Maurizio. 1999. «Measurement processes are software, too». *The Journal of Systems and Software*. 49. 17–31.

MÜNCH, Jürgen, HEIDRICH, Jens. 2004. «Software project control centers : Concepts and approaches». *The Journal of Systems and Software*. 70. 3–19.

NOLL, John, ATKINSON, Darren C. 2003. «Comparing Extreme Programming to traditional development for student projects : A case study». *Extreme Programming and Agile Processes in Software Engineering*. Berlin : Springer. P. 372–374.

OBJECT MANAGEMENT GROUP. 2002. «Meta object facility (MOF) specification». [En ligne]. http ://www.omg.org/ technology/documents/formal/mof.htm (Page consultée le 21 mars 2004).

OPEN CONSORTIUM. 2002. «OPEN : Object-oriented process, environment and notation». [En ligne]. http ://www.open.org.au (Page consultée le 21 mars 2004).

OSTERWEIL, Leon J. 1987. «Software processes are software too». *Proceedings, 9th International Conference on Software Engineering*. New York : IEEE Computer Society Press. P. 2–13.

OSTWALD, Philip F. 1992. «Cost estimating». *Handbook of Industrial Engineering*. Sous la direction de Gavriel Salvendy. 2nd ed. New York : Wiley. P. 1263–1288.

PALMER, Stephen R., FELSING, John M. 2002. *A Practical Guide to Feature-Driven Development*. Upper Saddle River : Prentice Hall PTR. 271 p.

PAULK, Mark C., CURTIS, Bill, CHRISSIS, Mary Beth, WEBER, Charles W. 1993. «Capability Maturity Model version 1.1». *IEEE Software*. 10:4. 18–27.

PRESSMAN, Roger S. 2000. *Software Engineering : A Practitioner's Approach*. 6th ed. Boston : McGraw Hill. 860 p.

RISING, Linda, JANOFF, Norman S. 2000. «The Scrum software development process for small teams». *IEEE Software*. 17:4. 26–32.

ROBILLARD, Pierre N. 1996. «Teaching software engineering through a project-oriented course». *Proceedings of the 9th Conference on Software Engineering Education (CSEE'96)*. Los Alamitos : IEEE Computer Society. P. 85–94.

ROBILLARD, Pierre N. 1998. «Measuring team activities in a process-oriented software engineering course». *Proceedings of the 11th Conference on Software*

Engineering Education and Training (CSEET'98). Los Alamitos : IEEE Computer Society. P. 90–101.

ROBILLARD, Pierre N., KRUCHTEN, Philippe, D'ASTOUS, Patrick. 2001. «YOOPEEDOO (UPEDU) : A process for teaching software process». *Proceedings of the 14th Conference on Software Engineering Education and Training (CSEET'01)*. Los Alamitos : IEEE Computer Society. P. 18–26.

ROBILLARD, Pierre N., KRUCHTEN, Philippe, D'ASTOUS, Patrick. 2003. *Software engineering process with the UP/Edu*. Boston : Addison Wesley. 346 p.

ROYCE, Winston W. 1970. «Managing the development of large software systems : Concepts and techniques». *1970 WESCON Technical Papers*. Los Alamitos : IEEE Computer Society. P. A/1–1 – A/1–9.

SEAMAN, Carolyn B. 1999. «Qualitative methods in empirical studies of software engineering». *IEEE Transactions on Software Engineering*. 25:4. 557–572.

SEAMAN, Carolyn B., BASILI, Victor R. 1998. «Communication and organization : An empirical study of discussion in inspection meetings». *IEEE Transactions on Software Engineering*. 24:7. 559–572.

SELBY, Richard W., PORTER, Adam A., SCHMIDT, Doug C., BERNEY, Jim. 1991. «Metric-driven analysis and feedback systems for enabling empirically guided software development». *Proceedings, 13th International Conference on Software Engineering*. Los Alamitos : IEEE Computer Society Press. P. 288–298.

SLAVICH, John. 2000. *Processus de quantification des attributs logiciels (PQAL) basé sur une méthode de prise de décisions multicritères*. 147 p. Mémoire de maîtrise en génie électrique, École Polytechnique de Montréal.

STANDARDS COORDINATING COMMITTEE OF THE COMPUTER SOCIETY OF THE IEEE. 1990. *IEEE Standard Glossary of Software Engineering Terminology*. New York : Institute of Electrical and Electronics Engineers. 83 p.

STANDISH GROUP. 2001. «Chaos chronicles version 2.0». 316 p.

STAPLETON, Jennifer. 1997. *DSDM, Dynamic Systems Development Method : The Method in Practice*. Harlow : Addison Wesley. 192 p.

TICHY, Walter F. 2000. «Hints for reviewing empirical work in software engineering». *Empirical Software Engineering*. 5:4. 309–312.

UMPHRESS, David A., HAMILTON, Jr., John A. 2002. «Software process as a foundation for teaching, learning, and accrediting». *Proceedings of the 15th Conference on Software Engineering Education and Training (CSEET'02)*. Los Alamitos : IEEE Computer Society. P. 160–169.

UNIVERSITY OF SOUTHERN CALIFORNIA. 1998. «COCOMO II model definition manual». [En ligne]. ftp ://ftp.usc.edu/pub/soft_engineering/ COCOMOII/cocomo97docs/modelman.ps.

WELLS, Don. 2003. «Don't solve a problem before you get to it». *IEEE Software*. 20:3. 44–47.

WILLIAMS, Laurie, KESSLER, Robert R., CUNNINGHAM, Ward, JEFFRIES, Ron. 2000. «Strengthening the case for pair programming». *IEEE Software*. 17:4. 19–25.

WILLIAMS, Laurie, UPCHURCH, Richard L. 2001. «In support of student pair-programming». *Technical Symposium on Computer Science Education, Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education*. New York : ACM Press. P. 327–331.

WILLIAMS, Sam. 2002. «A unified theory of software evolution». [En ligne]. http ://www.salon.com/tech/feature/ 2002/04/08/lehman/index.html (Page consultée le 21 mars 2004).

Annexe I

Calculation of the 2D plots of effort

Consider x, y and z to represent relative effort expended on coding, V&V and engineering activities respectively. Let $B = \left\{\vec{\imath}, \vec{\jmath}, \vec{k}\right\}$ be a base for $\mathbb{R}^3$. Since the sum of relative effort at a given time is 100%, we obtain for each vector

$$\left[\vec{E}\right]_B = \begin{bmatrix} x \\ y \\ z \end{bmatrix} :$$

$$x + y + z - 1 = 0 \tag{I.1}$$

which represents the equation of a plane in a three-dimensional space. Such a plane can be represented in a two-dimensional space if rotated properly. We may define the following vectors $\vec{\imath}$, $\vec{\jmath}$ and $\vec{k}$ as follows:

$$\left[\vec{\imath'}\right]_B = \begin{bmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \\ 0 \end{bmatrix} \tag{I.2}$$

$$\left[\vec{j}\right]_B = \begin{bmatrix} 1/\sqrt{6} \\ 1/\sqrt{6} \\ -2/\sqrt{6} \end{bmatrix} \tag{I.3}$$

$$\left[\vec{k'}\right]_B = \begin{bmatrix} 1/\sqrt{3} \\ 1/\sqrt{3} \\ 1/\sqrt{3} \end{bmatrix} \tag{I.4}$$

Since $\|\vec{i'}\| = \|\vec{j'}\| = \|\vec{k'}\| = 1$ and $\vec{i'} \times \vec{j'} = \vec{k'}$, we can conclude that $\vec{i'}$, $\vec{j'}$ and $\vec{k'}$ form three linearly independent unit vectors, and that, therefore, it is possible to express $\vec{E}$ in terms of the new base $B' = \left\{\vec{i'}, \vec{j'}, \vec{k'}\right\}$. Let:

$$P = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{6} & 1/\sqrt{3} \\ -1/\sqrt{2} & 1/\sqrt{6} & 1/\sqrt{3} \\ 0 & -2/\sqrt{6} & 1/\sqrt{3} \end{bmatrix} \tag{I.5}$$

be the change of coordinate matrix from B' to B. Therefore

$$\left[\vec{E}\right]_{B'} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = P^{-1}\left[\vec{E}\right]_B$$

$$= \begin{bmatrix} (x-y)/\sqrt{2} \\ (x+y-2z)/\sqrt{6} \\ (x+y+z)/\sqrt{3} \end{bmatrix} = \begin{bmatrix} (x-y)/\sqrt{2} \\ (x+y-2z)/\sqrt{6} \\ 1/\sqrt{3} \end{bmatrix}. \tag{I.6}$$

We used equation I.6 to draw each 2D plot diagram (dropping z' which is constant). Any other relative effort mix can be represented in the same way in a similar diagram.

Annexe II

Contenu du support CD fourni

Le disque compact qui accompagne ce mémoire contient trois articles de conférence auquel l'auteur a contribué. Les trois articles sont en format PDF et peuvent être visualisés notamment à l'aide du logiciel Adobe® Reader®, version 6,0.

Les articles contenus sont les suivants : Germain, Dulipovici et Robillard (2002a); Germain, Robillard et Dulipovici (2002b); Germain et Robillard (2003). Les noms des fichiers correspondent aux identificateurs apparaissant à la bibliographie.